Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

The s-semantics approach: theory and applications

A. Bossi, M. Gabbrielli, G. Levi, M. Martelli

Computer Science/Department of Software Technology

CS-R9408 1994

The s-semantics approach: theory and applications

A. Bossi
Dipartimento di Matematica Pura ed Applicata
Università di Padova, Via Belzoni 7, 35131 Padova, Italy
isa@zenone.unipd.it

M. Gabbrielli[1]
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
gabbri@cwi.nl

G. Levi
Dipartimento di Informatica
Università di Pisa, Corso Italia 40, 56100 Pisa, Italy
levi@di.unipi.it

M. Martelli
Dipartimento di Informatica e Scienze dell'Informazione
Università di Genova, Viale Benedetto XV 3, 16132 Genoa, Italy
martelli@disi.unige.it

Abstract

The paper is a general overview of an approach to the semantics of logic programs whose aim is finding notions of models which really capture the operational semantics, and are therefore useful for defining program equivalences and for semantics-based program analysis. The approach leads to the introduction of extended interpretations which are more expressive than Herbrand interpretations. The semantics in terms of extended interpretations can be obtained as a result of both an operational (top-down) and a fixpoint (bottom-up) construction. It can also be characterized from the model-theoretic viewpoint, by defining a set of extended models which contains standard Herbrand models. We discuss the original construction modeling computed answer substitutions, its compositional version and various semantics modeling more concrete observables. We then show how the approach can be applied to several extensions of positive logic programs. We finally consider some applications, mainly in the area of semantics-based program transformation and analysis.

1991 Mathematics Subject Classification: 68N17, 68Q55.
CR Categories: D.1.6, D.3.1, F.3.2, F.4.1, I.2.3.
Keywords and Phrases: logic programming, semantics.
Note: To appear in the *Journal of Logic Programming.*

[1]On leave from Dipartimento di Informatica, Università di Pisa, Corso Italia 40, Pisa, Italy.

1 Introduction

1.1 Denotations as syntactic objects

The paper considers an approach to the semantics of logic programs which leads to *denotations consisting of (equivalence classes of) syntactic objects*. There are two main motivations for using syntactic domains. Namely,

- syntactic domains make possible the definition of program denotations which capture various computational aspects in a goal-independent way. These aspects include observable properties such as
  - *computed answers*, which are modeled by sets of non-ground atoms or unit clauses [47] (see section 3),
  - *call patterns*, which are modeled by sets of binary clauses [61] (see section 5.3.2),
  - *resultants*, which are modeled by sets of clauses [59] (see section 5.3).

  Goal-independence is the key issue. It means that denotations are defined by collecting the observable properties starting with the most general atomic goals and that they give a complete characterization of the program behavior for any goal.
- syntactic domains make possible the definition of a unique denotation in cases where there exists no unique representative Herbrand model. Examples are
  - the *compositional semantics for positive logic programs* [62, 63, 22, 21], whose domains are sets of clauses (see section 4),
  - the *semantic kernel for normal logic programs* [43, 75], whose domains are sets of negative normal clauses (see section 6.3),
  - the *model state semantics for disjunctive logic programs* [98, 90], whose domains are sets of positive disjunctive ground clauses (see section 6.2).

The overall approach is called in this paper *the s-semantics approach* after the s-semantics [47], which was the first example of a semantic construction featuring some of the above properties. By no means we imply that all the denotations we consider are extensions of the original s-semantics.

1.2 Why a new semantics

According to a popular view of logic programming, the problem of the semantics (of definite Horn clauses) was solved once and for all by logicians before logic programming was even born. Namely, the only three important concepts are the *program* itself, the *intended interpretation* (declarative semantics) and the *theorem prover* (operational semantics). The program is a logic theory. The declarative semantics formalizes the application the program is trying to capture. It is an interpretation in the conventional logic sense and a model of the program. Finally, the theorem prover is a proof procedure which must be sound (and complete) with respect to the declarative semantics. Is that really *all there is to it*?

The above view is appealing but too simple minded to capture the difference between theorem proving and programming. In fact, it applies to any formal system for which there exists a sound and complete theorem prover. Theorem proving becomes logic programming when we restrict the class of theories so as to obtain a declarative semantics (a unique model) and

a proof procedure similar to the denotational and the operational semantics of conventional programming languages. This is exactly what van Emden and Kowalski did for definite Horn clauses in their seminal paper [109], where the proof procedure was SLD-resolution and the model was the least Herbrand model. *The* semantics is then a mathematical object which is defined in model-theoretic terms and which can be *computed* by a top-down construction (the success set) and by a bottom-up construction (the least fixpoint of the immediate consequences operator). Why shouldn't we be happy with this solution?

The answer can be found if we first consider a different and more basic question. What is a semantics used for? The first *application* of any semantics is to help understanding the meaning of programs. Other useful applications include areas such as program transformation and program analysis. One can argue that tens of thousands of logic programmers were really helped by the declarative understanding of their programs. One can also argue that semantics-based program transformation and analysis do require deeper results and more elaborate theories, but still only using basically the above mentioned simple and straightforward semantics. The above arguments can become more technical only if we understand which is the basic semantic property of such formal activities as program transformation and analysis. The answer is *program equivalence*, i.e. program understanding is based on our ability to detect when two programs cannot be distinguished by looking at their behaviors.

1.3 Program equivalences and observables

Defining an equivalence on programs $\approx$ and a formal semantics $\mathcal{S}(P)$ are two strongly related tasks. A semantics $\mathcal{S}(P)$ is *correct* w.r.t. $\approx$, if $\mathcal{S}(P_1) = \mathcal{S}(P_2)$ implies $P_1 \approx P_2$. The question about the adequacy of the van Emden and Kowalski's semantics can then be rephrased as follows. Is that semantics correct w.r.t. a "natural" notion of program equivalence? This in turn raises the problem of choosing a suitable notion of equivalence.

Equivalences can be defined by using logical arguments only. One can use model-theoretic properties, such as the set of models, the set of logical consequences or the least Herbrand model, and proof-theoretic properties, such as the set of derivable atoms. A systematic comparison of several program equivalences has been worked out in [91]. In particular, [91] shows the relations between equivalences based on purely logical properties and equivalences induced on programs by more "operational" aspects. For example, subsumption equivalence of two programs is shown to correspond to the equality of their T_P operators. Equivalences based on correct answer substitutions have also been studied in [29]. However, these formalizations are not completely satisfactory since they do not consider an important class of program equivalences, which cannot be described by purely (standard first-order) logical notions. This is the class of equivalences based on what we can observe from a computation.

One important aspect of the formalization of program execution, in addition to the inference rules which specify how derivations are made, is the concept of *observable*, i.e. the property we observe in a computation. In logic programs we can be interested in different observable properties such as successful derivations, finite failures, computed answer substitutions, partial computed answer substitutions, finite sets of solutions, etc. A given choice of the observable X induces an *observational equivalence* $\approx_X$ on programs. Namely $P_1 \approx_X P_2$ iff P_1 and P_2 are observationally indistinguishable according to X. For example, if s denotes *successful derivations*, $P_1 \approx_s P_2$ iff for any goal G, G is refutable in P_1 iff it is refutable in P_2. This observable is adequate to characterize a theorem prover, yet it is definitely too abstract to capture the essence of logic programming, i.e. the ability to compute answers. The most adequate observable is therefore *computed answers* (denoted by c). $P_1 \approx_c P_2$ iff for any goal G, G has the same

(up to renaming) computed answers in P_1 and in P_2.

As first shown in [47], *the van Emden and Kowalski's semantics is not correct w.r.t. to the observational equivalence based on computed answer substitutions.* Namely, there exist programs which have the same least Herbrand model, yet compute different answer substitutions. When trying to understand the meaning of programs, when analyzing and transforming programs, this semantics cannot be taken as the reference semantics. This is the reason why the need for a different formal semantics was recognized by many authors, giving rise to several new definitions [30, 49, 113, 42]. The need for better semantics was also recognized in the case of semantics-based abstract interpretation [94] and transformation [76].

1.4 Compositionality

In addition to the problem related to modeling the computed answers observational equivalences, there exists another problem with the least Herbrand model semantics. Namely a very important property, i.e. *compositionality*, does not hold. Compositionality has to do with a (syntactic) program composition operator $\circ$, and holds when the semantics of the compound construct $C_1 \circ C_2$ is defined by (semantically) composing the semantics of the constituents C_1 and C_2. In the case of logic programs, the construct which raises a compositionality problem is the *union* of clauses. The related property is called $\cup$-*compositionality*. $\cup$-compositionality is interesting both for theoretical and for practical (i.e. the definition of semantics for modular versions of logic programs) purposes. When also composition of programs is taken into account, for a given observable property we obtain different equivalences depending on which kind of program composition we consider. Given an observable X and a program composition operator $\circ$, the induced congruence $\approx_{(\circ,X)}$ is defined as follows. $P_1 \approx_{(\circ,X)} P_2$ iff for any program Q, $P_1 \circ Q \approx_X P_2 \circ Q$, (i.e. iff P_1 and P_2 are observationally indistinguishable under any possible context allowed by the composition operator).

1.5 Plan of the paper

In the next section we describe the general approach. In section 3 we consider the original s-semantics [47, 48], which is the first (non-compositional) semantics of positive logic programs correct w.r.t. computed answers. Compositionality is discussed in section 4, while in section 5 we consider semantics modeling other observables, such as finite failures and resultants. Section 6 discusses the application of the approach to several extensions of positive logic programs, including *constraint logic programs*, *disjunctive logic programs*, *normal logic programs*, *constructive negation*, *structured logic programs with inheritance* and *Prolog programs*. Finally, section 7 shows some applications of the approach, in the areas of program transformation, semantics-based analysis and metaprogramming.

1.6 Preliminaries

The reader is assumed to be familiar with the terminology of and the basic results in the semantics of logic programs [88, 3]. Let $\mathcal{L}$ be the first order language defined by the signature S consisting of a set C of *data constructors*, a finite set P of *predicate symbols*, a denumerable set V of *variable symbols*. Let $\mathcal{L}'$ be the language defined by C', P' and V and $\mathcal{L}$ be the language defined by C, P and V. $\mathcal{L}'$ is an *extension* of $\mathcal{L}$ if $C \subseteq C'$ and $P \subseteq P'$. When the language will be subscripted by the program, as in $\mathcal{L}_P$, the signature will be the one defined by the symbols occurring in the program P. Otherwise a given signature S is assumed.

Let T be the set of terms built on C and V. Variable-free terms are called *ground*. A substitution is a mapping $\vartheta : V \to T$ such that the set $D(\vartheta) = \{X \mid \vartheta(X) \neq X\}$ (*domain* of ϑ) is finite. If $W \subset V$, we denote by $\vartheta_{|W}$ the *restriction* of ϑ to the variables in W, i.e. $\vartheta_{|W}(Y) = Y$ for $Y \notin W$. Moreover if E is any syntactic object, we use the abbreviation $\vartheta_{|E}$ to denote $\vartheta_{|Var(E)}$. ε denotes the empty substitution. The *composition* $\vartheta\sigma$ of the substitutions ϑ and σ is defined as the functional composition, i.e. $\vartheta\sigma(x) = \sigma(\vartheta(x))$. A *renaming* is a substitution ρ for which there exists the inverse ρ^{-1} such that $\rho\rho^{-1} = \rho^{-1}\rho = \varepsilon$. The preordering $\leq$ (more general than) on substitutions is such that $\vartheta \leq \sigma$ iff there exists ϑ' such that $\vartheta\vartheta' = \sigma$. The result of the application of the substitution ϑ to a term t is an *instance* of t denoted by $t\vartheta$. We define $t \leq t'$ (t is more general than t') iff there exists ϑ such that $t\vartheta = t'$. A substitution ϑ is *grounding* for t if $t\vartheta$ is ground. The relation $\leq$ is a preorder. $\approx$ denotes the associated equivalence relation (*variance*). A substitution ϑ is a *unifier* of terms t and t' if $t\vartheta = t'\vartheta$. $mgu(t_1, t_2)$ denotes any idempotent *most general unifier* of t_1 and t_2. All the above definitions can be extended to other syntactic objects in the obvious way.

A *literal* L is an object of the form $p(t_1, \ldots, t_n)$ (*atom*) or $\neg p(t_1, \ldots, t_n)$ (*negative literal*), where $p \in P$, $t_1, \ldots, t_n \in T$ and "$\neg$" denotes negation. A *clause* is a formula of the form $H : -L_1, \ldots, L_n$ with $n \geq 0$, where H (the *head*) is an atom and $L_1, \ldots, L_n$ (the *body*) are literals. "$: -$" and "," denote logic implication and conjunction respectively, and all variables are universally quantified. A *definite clause* is a clause whose body contains atoms only. If the body is empty the clause is a *unit clause*. A *normal program* is a finite set of clauses $P = \{c_1, \ldots, c_n\}$. A *positive program* is a finite set of definite clauses. A *normal (positive) goal* is a formula $L_1, \ldots, L_m$, where each L_i is a literal (atom).

A *Herbrand interpretation* I for a program P is a subset of the *Herbrand base* B (the set of all ground atoms). The intersection $M(P)$ of all the Herbrand models of a positive program P is a model (least Herbrand model). $M(P)$ is also the least fixpoint $T_P \uparrow \omega$ of a continuous transformation T_P (*immediate consequences operator*) on Herbrand interpretations. The *ordinal powers* of a generic monotonic operator T_P on a complete lattice $(D, \leq)$ with bottom $\bot$ are defined as usual, namely $T_P \uparrow 0 = \bot$, $T_P \uparrow (\alpha + 1) = T_P(T_P \uparrow \alpha)$ for α successor ordinal and $T_P \uparrow \alpha = lub(\{T_P \uparrow \beta \mid \beta < \alpha\})$ if α is a limit ordinal. If G is a positive goal, $G \overset{\vartheta}{\leadsto}_{P,R} B_1, \ldots, B_n$ denotes an SLD-derivation of $B_1, \ldots, B_n$ from the goal G in the program P which uses the selection rule R and such that ϑ is the composition of the mgu's used in the derivation. $G \overset{\vartheta}{\longmapsto}_P \Box$ denotes the SLD-refutation of G in the program P with computed answer substitution ϑ. A computed answer substitution is always restricted to the variables occurring in G.

We will denote by $\tilde{X}$ and $\tilde{t}$ a tuple of distinct variables and a tuple of terms respectively, while $\tilde{B}$ will denote a (possibly empty) conjunction of atoms. For any set A, A^* denotes the set of finite sequences of elements of A. :: will denote concatenation of sequences and λ is the empty sequence.

2 The *s*-semantics approach

The aim of the *s*-semantics approach [82, 56, 52, 58] is modeling the observable behaviors (possibly in a compositional way) for a variety of logic languages. The approach is based on the idea of choosing (equivalence classes of) sets of clauses as semantic domains. The denotations are then defined by *syntactic* objects, as in the case of Herbrand interpretations. Denotations (called *π-interpretations*) are not interpretations in the conventional mathematical logic sense. As in the case of the van Emden and Kowalski semantics, denotations can be computed both by

a top-down construction (a success set) and by a bottom-up construction (the least fixpoint of suitable continuous immediate consequences operators on π-interpretations). The link between the top-down and the bottom-up constructions is given by an *unfolding operator* [82, 83]. The equivalence proofs can be stated in terms of simple properties of the unfolding and the immediate consequences operators [41].

It is worth noting that the aim of the approach is not defining a new notion of model. We are simply unhappy with the traditional declarative semantics, because it characterizes the logical properties only and we look for new notions of program denotation useful from the programming point of view. A satisfactory solution to the simple case of positive logic programs is needed to gain a better understanding of more practical languages, such as real Prolog and its purely declarative counterparts.

We show our construction in a language independent way by considering three separate steps, which roughly correspond to the three standard semantics of logic programs [109, 88, 3]. The first step is related to the operational semantics and leads to the definition of the structure of π-interpretations. The second step is concerned with the fixpoint semantics. The third and final step is concerned with the definition of π-models.

2.1 Observable properties and π-interpretations

The operational semantics is usually given by means of a set of inference rules which specify how derivations are made, and by defining a proper notion of observable. Consider for example positive logic programs with no composition and computed answer substitutions as observable. As we will show in section 3, the denotation of a program is a set of non-ground atoms, which can be viewed as a possibly infinite program. This is just an instance of a more general property of denotations within our approach. Namely denotations are possibly infinite programs and semantic domains are made of syntactic objects. The amount of syntax which is needed in the semantic domains depends on the observable and on the composition. For example, in the computed answer substitutions semantics, the syntactic construct of variables is added to the Herbrand domain. When considering $\cup$-composition also, non-ground unit clauses are not sufficient any longer and more general clauses are needed (see section 4). Note that the approach is feasible only if the language syntax is powerful enough to express its own semantics. Since we have syntactic objects in the semantic domain, we need an equivalence relation in order to abstract from irrelevant syntactic differences. In the above considered example, this relation is variance. If the equivalence is accurate enough the semantics is fully abstract.

Herbrand interpretations are generalized in our setting by π-interpretations which are possibly infinite sets of equivalence classes of clauses from the semantics domain. The operational semantics of a program P is then a π-interpretation I, which has the following property. P and I are observationally equivalent with respect to any goal G. This is the property which allows us to state that the semantics does indeed capture the observable behavior.

2.2 Fixpoint semantics and unfolding

The aim of the second phase is the definition of a fixpoint semantics equivalent to the previously defined operational semantics. This can be achieved by the following steps.

- The set of π-interpretations is organized in a lattice $(\Im, \sqsubseteq)$ based on a suitable partial order relation $\sqsubseteq$, which in most cases is set inclusion.

- An immediate consequences operator T_P^π is defined and proved monotonic and continuous on $(\Im, \sqsubseteq)$. This allows us to define the fixpoint semantics $\mathcal{F}(P)$ for P as $\mathcal{F}(P) = T_P^\pi \uparrow \omega$.
- The fixpoint semantics $\mathcal{F}$ is proved equivalent to the operational semantics. If this equivalence holds, the immediate consequences operator T_P^π models the observable properties and may be used for bottom-up program analysis.

Concise and elegant equivalence proofs can be obtained by introducing the intermediate notion of *unfolding semantics* $\mathcal{U}$ [82, 83]. Unfolding is a well known program transformation rule which allows us to replace procedure calls by procedure definitions. The unfolding of the clauses of program P using the procedure definitions in program I is denoted by $unf_P(I)$.

The unfolding and the operational semantics are strongly related, since they are based on the same inference rule (applied to clauses and goals respectively). The unfolding semantics $\mathcal{U}(P)$ is obtained as the limit of the unfolding process. If the unfolding rule preserves the observable properties, $\mathcal{U}(P)$ is equivalent to the operational semantics $\mathcal{O}(P)$ which is a π-interpretation and therefore a program. This shows that the statement "the language syntax is powerful enough to express its own semantics" can be rephrased as "the language is closed under unfolding".

On the other side, the unfolding operator unf_P is strongly related to the immediate consequences operator T_P^π. For example, in many cases, given a π-interpretation I, the relation $T_P^\pi(I) = unf_P(I)$ holds. The proof of equivalence between $\mathcal{U}(P)$ and $\mathcal{F}(P)$ can be based on such a relation. In particular the equivalence immediately holds for those immediate consequences operators which are *compatible with* the unfolding rule [41]. The above relations suggest a methodology to obtain the immediate consequences operator by first defining the unfolding operator, which is easier to define because of its strong relation to the operational semantics.

2.3 Model-theoretic semantics

Let us first note that the original model-theoretic view of the s-semantics [47] was based on ad-hoc notions of s-truth and s-model. The notion of π-model, first introduced in [48], fixes the above problem, by viewing a denotation just as a syntactic notation for a set of Herbrand interpretations. $\mathcal{H}(I_P)$ denotes the set of all the Herbrand interpretations represented by I_P. For instance, in positive logic programs, the operational semantics $\mathcal{O}(P)$ is a set of non-ground atoms and $\mathcal{H}(\mathcal{O}(P))$ is the set containing the least Herbrand model of $\mathcal{O}(P)$. In general, our aim is finding a notion of π-model such that $\mathcal{O}(P)$ and $\mathcal{F}(P)$ are π-models and every Herbrand model is a π-model. This can be obtained by the following definition.

Definition 2.1 *Given a program P and a π-interpretation I, I is a* π-model *of P iff P is true in all the Herbrand interpretations in $\mathcal{H}(I)$.*

As we will show in the following, the model intersection property does not hold in general for π-models. This is due to the fact that set inclusion does not adequately correspond to the intended meaning of π-interpretations. Namely the information of a π-interpretation I_1 may be contained in I_2 without I_1 being a subset of I_2. In general, we look for a partial order $\preceq$ modeling the meaning of π-interpretations, such that $(\Im, \preceq)$ is a complete lattice and the greatest lower bound of a set of π-models is a π-model. According to the last property there exists a least π-model, which, as we will see in the following, is the least Herbrand model. It is worth noting that the most expressive π-model $\mathcal{O}(P)$ is a non-minimal π-model.

3 Positive logic programs

In this section we consider the original s-semantics[2], which is a non-compositional semantics for positive programs. Compositions will be considered in section 4.

3.1 Top-down semantics and π-interpretations

The first observable we consider is the *computed answer substitutions* which induces the following program equivalence $\simeq$. Other observable properties (and therefore different semantics) will be considered in Section 5.

Definition 3.1 *Let P_1, P_2 be positive programs. $P_1 \simeq P_2$ if for every positive goal G, $G \stackrel{\vartheta}{\longmapsto}_{P_1} \Box$ iff $G \stackrel{\vartheta'}{\longmapsto}_{P_2} \Box$ and $\vartheta = (\vartheta'\rho)_{|G}$, where ρ is a renaming.*

The above observable is captured by the following operational semantics. Recall that $\tilde{X}$ denotes a tuple of distinct variables.

Definition 3.2 (Computed answer substitutions semantics, s-semantics) *[47] Let P be a positive program.*

$$\mathcal{O}(P) = \{A \mid \begin{array}{l} \exists \tilde{X} \in V,\ \exists \vartheta, \\ p(\tilde{X}) \stackrel{\vartheta}{\longmapsto}_P \Box, \\ A = p(\tilde{X})\vartheta \end{array} \}$$

In order to model $\mathcal{O}(P)$ the usual Herbrand base has to be extended to the set of all the (possibly non-ground) atoms modulo variance.

Definition 3.3 *Let $\mathcal{B}$ be the quotient set of all the atoms w.r.t. variance. A π-interpretation is any subset of $\mathcal{B}$.*

In the following $\mathcal{O}(P)$ will then be formally considered as a subset of $\mathcal{B}$. Moreover, we will denote the equivalence class of an atom A by A itself. Note that π-interpretations of definition 3.3 are not Herbrand interpretations, yet are interpretations defined on the Herbrand universe. These interpretations were called *canonical realizations* in [100, 79].

Theorem 3.4 shows that $\mathcal{O}$ actually models computed answer substitutions and that it is *fully abstract*, since $P_1 \simeq P_2$ implies $\mathcal{O}(P_1) = \mathcal{O}(P_2)$.

Theorem 3.4 *[47] Let P_1, P_2 be positive programs. $P_1 \simeq P_2$ iff $\mathcal{O}(P_1) = \mathcal{O}(P_2)$.*

The following theorem asserts that the observable behavior of any (possibly conjunctive) goal can be derived from $\mathcal{O}(P)$, i.e. from the observable behaviors of atomic goals of the form $p(\tilde{X})$. This property is a kind of AND-compositionality. Similar theorems will be shown to hold for all the semantics defined according to the s-semantics style. This is also the key property which allows us to use abstractions of the semantics for goal independent abstract interpretation.

Theorem 3.5 *[47] Let P be a positive program and $G = G_1, \ldots, G_n$ be a positive goal. Then $G \stackrel{\vartheta}{\longmapsto}_P \Box$ iff there exist (renamed apart) atoms $A_1, \ldots, A_n \in \mathcal{O}(P)$ and a renaming ρ such that $\vartheta = (\gamma\rho)_{|G}$ where $\gamma = mgu((A_1, \ldots, A_n), (G_1, \ldots, G_n))$.*

[2] s stands for "subset interpretations" used in [47] as semantic domains and contrasted to "closed interpretations" used to define the so-called c-semantics (see Definition 3.6).

Theorem 3.5 shows that $\mathcal{O}(P)$ provides a denotation which can actually be used to simulate the program execution for any goal $G = G_1, \ldots, G_n$. Namely the answer substitutions for G can be determined by "executing G in $\mathcal{O}(P)$", i.e. by computing a most general unifier of $G_1, \ldots, G_n$ and $A_1, \ldots, A_n$, where the A_i's are renamed apart variants of atoms in $\mathcal{O}(P)$.

Let us consider now the *success set* and the *atomic logical consequences semantics* formally defined as follows.

Definition 3.6 *Let P be a positive program.*
(success set) $\mathcal{O}_1(P) = \{A \mid A \text{ is ground and } A \stackrel{\varepsilon}{\longmapsto}_P \Box\}$
(atomic logical consequences semantics) $\mathcal{O}_2(P) = \{A \mid A \stackrel{\varepsilon}{\longmapsto}_P \Box\}$

Note that the semantic domain of $\mathcal{O}_1$ is the usual Herbrand base, i.e. the set of all the ground atoms. Note also that $\mathcal{O}_2$ is the semantics considered in [30, 42, 62] and called c-semantics in [47]. We will now compare the three semantics on an example.

Example 3.7 *Consider the programs P_1 and P_2 on the signature S, defined by $C = \{a\backslash 0, f\backslash 1\}$.*

$$P_1 = \{\ p(a).\quad p(X).\quad q(f(a)).\} \qquad P_2 = \{\ p(X).\quad q(f(a)).\}$$

$$\begin{array}{lllllll}
\mathcal{O}(P_1) = & \{q(f(a)), & p(X), & p(a) & & & \} \\
\mathcal{O}(P_2) = & \{q(f(a)), & p(X) & & & & \} \\
\mathcal{O}_1(P_1) = \mathcal{O}_1(P_2) = & \{q(f(a)), & & p(a), & & p(f(a)), & \ldots\} \\
\mathcal{O}_2(P_1) = \mathcal{O}_2(P_2) = & \{q(f(a)), & p(X), & p(a), & p(f(X)), & p(f(a)), & \ldots\}
\end{array}$$

Note that $P_1 \simeq P_2$ does not hold, since the goal $p(X)$ computes different answer substitutions in P_1 and in P_2. Note also that the denotations defined by $\mathcal{O}$ are finite, while those computed by both $\mathcal{O}_1$ and $\mathcal{O}_2$ are infinite.

Example 3.7 shows that the three semantics are different. Indeed, if we denote by $\equiv_i$ the program equivalence induced by $\mathcal{O}_i$, $i = 1, 2$, the following (strict) inclusion holds [47, 48]. $\simeq \subseteq \equiv_2 \subseteq \equiv_1$, i.e. $\simeq$ is finer than $\equiv_2$, and $\equiv_2$ is finer than $\equiv_1$. This shows that the success set semantics is not correct with respect to computed answers. Moreover the correctness cannot be achieved by just using interpretations consisting of sets of non-ground atoms. In fact also the c-semantics does not correctly model the computed answers.

Let I be a π-interpretation. If $[I]$ denotes the set of ground instances of the atoms in I, $[I]$ is clearly a Herbrand interpretation. The following theorem relates the s-semantics to the success set (and therefore to the least Herbrand model).

Theorem 3.8 *[47] If P is a positive program, then $\mathcal{O}_1(P) = [\mathcal{O}(P)]$.*

We have shown that the success set semantics does not correctly model the computed answers. One could still think that this is not the case in most reasonable logic programs. Which is the class of positive programs for which the success set is correct with respect to computed answers? This is clearly the case for the class of programs for which the s-semantics and the least Herbrand model semantics do coincide. Theorem 3.10 shows that this is exactly the class of *language independent* programs as defined in [37].

Definition 3.9 *[37] A program P with underlying language $\mathcal{L}_P$ is* language independent *iff, for any extension $\mathcal{L}'$ of $\mathcal{L}_P$, its least $\mathcal{L}'$-Herbrand model is equal to its least $\mathcal{L}_P$-Herbrand model.*

Theorem 3.10 *[85] Let P be a program. Then P is language independent iff $\mathcal{O}(P) = \mathcal{O}_1(P)$.*

A program P belongs to this class only if any goal in P returns ground answers. It is therefore essentially the class of *allowed* positive programs [88] and does not contain any program able to compute partial data structures.

The success set semantics does not need to be the same as the s-semantics in order to be correct with respect to computed answers, rather it needs to be isomorphic. The class of programs for which this property holds has been studied in [5].

Another related useful property of the s-semantics is its independence from the language. This means that the denotation defined by $\mathcal{O}$ is not affected by the choice of the language signature. The language signature affects the domain of π-interpretations $\mathcal{B}$. Since $\mathcal{O}(P)$ is a subset of $\mathcal{B}$ it might also be affected. Therefore, let us denote by $\mathcal{O}^{\mathcal{L}}(P)$ the denotation for a given language $\mathcal{L}$. If $\mathcal{L}_P$ is the language underlying program P, the following theorem shows the language independence property. Note that the same property does not hold for other variable-based semantics, such as those in [30, 49].

Theorem 3.11 *[85] If P is a positive program, then $\mathcal{O}^{\mathcal{L}_P}(P) = \mathcal{O}^{\mathcal{L}'}(P)$ for any extension $\mathcal{L}'$ of $\mathcal{L}_P$.*

As we will show in section 7.4, this is the key property which makes the s-semantics adequate to formalize metaprogramming with the non-ground metalevel representation of object level variables.

3.2 Fixpoint semantics

We will now introduce an immediate consequences operator T_P^π on π-interpretations whose least fixpoint will be shown to be equivalent to the computed answer substitutions semantics $\mathcal{O}(P)$.

Lemma 3.12 *The set of all π-interpretations $(\Im, \subseteq)$ is a complete lattice.*

Definition 3.13 *[47] Let P be a positive program and I be a π-interpretation.*

$$T_P^\pi(I) = \{A \in \mathcal{B} \mid \begin{array}{l} \exists C = A' : -B_1, \ldots, B_n \in P, \\ \exists\ B'_1, \ldots, B'_n \textit{ variants of atoms in } I \textit{ and renamed apart}, \\ \exists \vartheta = mgu((B_1, \ldots, B_n), (B'_1, \ldots, B'_n)) \textit{ and } A = A'\vartheta \end{array} \}$$

Note that T_P^π is different from the standard T_P operator [109] in that it derives instances of the clause heads by *unifying* the clause bodies with atoms in the current π-interpretation, rather than by taking all the possible ground instances. In other words T_P^π defines a bottom-up inference rule (hyper-resolution) based on the same rule (unification) which is used by the top-down SLD-resolution. The following theorem allows us to define a fixpoint semantics for positive logic programs.

Theorem 3.14 *[47] The T_P^π operator is continuous on $(\Im, \subseteq)$. Then there exists the least fixpoint $T_P^\pi \uparrow \omega$ of T_P^π.*

Definition 3.15 *[47] The fixpoint semantics of a positive program P is defined as $\mathcal{F}(P) = T_P^\pi \uparrow \omega$.*

It is worth noting that, since any program P is a finite set of clauses, all the finite fixpoint approximations $T_P^\pi \uparrow n, n \leq \omega$ are finite. The T_P^π operator can then effectively be used for the construction of bottom-up proofs.

The equivalence between $\mathcal{F}(P)$ and $\mathcal{O}(P)$ is proved by introducing the unfolding semantics.

Definition 3.16 *[82, 83] Let P and Q be positive programs. Then the unfolding of P w.r.t. Q is defined as*

$$unf_P(Q) = \{(A : -\tilde{L}_1, \ldots, \tilde{L}_n)\vartheta \mid \begin{array}{l} \exists A : -B_1, \ldots, B_n \in P, \\ \exists B_i' : -\tilde{L}_i \in Q, i = 1, \ldots, n, \\ \textit{renamed apart, such that} \\ \vartheta = mgu((B_1, \ldots, B_n), (B_1', \ldots, B_n'))\} \end{array}$$

The unfolding rule can be applied to any atom in a clause and preserves the operational semantics, i.e. the language is closed under unfolding. Therefore it is possible to define the immediate consequences operator in terms of the unfolding rule. Theorem 3.21 was proved in [83]. An alternative proof is given in [41] by using lemma 3.20. A direct proof of $\mathcal{F}(P) = \mathcal{O}(P)$ was first given in [47].

Definition 3.17 *[83, 41] Let P be a positive program. Then we define the collection of programs*

$$\begin{array}{l} P_0 = P \\ P_i = unf_{P_{i-1}}(P), \ i = 1, 2, \ldots \end{array}$$

and the collection of π-interpretations $I_i(P) = \{A \mid A \in \mathcal{B} \text{ and } A \in P_i\}$. The unfolding semantics $\mathcal{U}(P)$ of the program P is defined as

$$\mathcal{U}(P) = \bigcup_{i=0,1,\ldots} I_i(P).$$

Theorem 3.18 (equivalence of unfolding and operational semantics) *[83, 41] Let P be a positive program. Then $\mathcal{U}(P) = \mathcal{O}(P)$.*

Definition 3.19 *[41] Let P, Q be positive programs. Then T_P^π is compatible with $unf_P(Q)$ iff $T^\pi_{unf_P(Q)}(\emptyset) = T_P^\pi(T_Q^\pi(\emptyset))$.*

Lemma 3.20 *[41] Let P, Q be positive programs. Then T_P^π is compatible with $unf_P(Q)$.*

Since T_P^π is compatible with the unfolding rule and $T_P^\pi(I) = unf_P(I)$ (by definition of the unfolding rule), then $T_P^\pi \uparrow (i+1) = T_{P_i}^\pi(\emptyset) = unf_{P_i}(\emptyset)$. Therefore,

Theorem 3.21 (equivalence of fixpoint and operational semantics) *[83, 41] Let P be a positive program. Then $\mathcal{F}(P) = \mathcal{U}(P) = \mathcal{O}(P)$.*

Theorem 3.21 shows that $\mathcal{F}(P)$ is the fully abstract semantics w.r.t. computed answer substitutions. The above equivalence between the top-down and the bottom-up semantics will hold for all our semantics, including the abstract versions used for program analysis. This makes available equivalent top-down and bottom-up proof methods.

3.3 Model-theoretic semantics

In order to define π-models according to definition 2.1 we have to specify the function $\mathcal{H}$ from π-interpretations to sets of Herbrand interpretations.

Definition 3.22 *[48] Let I be a π-interpretation. Then $\mathcal{H}(I) = \{[I]\}$ where $[I]$ is the set of ground instances of atoms in I or, equivalently, the least Herbrand model of I.*

Proposition 3.23 *[48] Let P be a program. Then every Herbrand model of P is a π-model of P. Moreover $\mathcal{O}(P), \mathcal{O}_1(P), \mathcal{O}_2(P)$ are π-models of P.*

The program P_2 of example 3.7 shows that the model intersection property does not hold any longer. In fact, $\mathcal{O}(P_2) \cap \mathcal{O}_1(P_2) \cap \mathcal{O}_2(P_2) = \{q(f(a))\}$ which is not a π-model of P_2. This is not surprising, since set theoretic operations do not adequately model the operations on non-ground atoms, which stand for all their ground instances. A more adequate partial order relation $\preceq$ on the set $\Im$ of π-interpretations was defined in [48].

Definition 3.24 *[48] Let I_1, I_2 be π-denotations. We define:*

- *$I_1 \leq_I I_2$ iff $\forall A_1 \in I_1$, $\exists\ A_2 \in I_2$ such that $A_2 \leq A_1$.*
- *$I_1 \preceq I_2$ iff ($I_1 \leq_I I_2$) and ($I_2 \leq_I I_1$ implies $I_1 \subseteq I_2$).*

$\preceq$ allows us to prove the following properties

- $(\Im, \preceq)$ is a complete lattice. $\mathcal{B}$ is the top element and $\emptyset$ is the bottom element.
- If $\mathbf{M}$ is a set of π-models of P, then $glb(\mathbf{M})$ is a π-model of P.
- The least π-model $\mathcal{M}(P) = glb(\{I \in \Im \mid I \text{ is a } \pi\text{-model of } P\})$ is the least Herbrand model.

It is worth noting that, according to proposition 3.23, the s-semantics $\mathcal{O}(P)$ is simply a non-ground representation of the least Herbrand model $\mathcal{O}_1(P)$. From the Herbrand models viewpoint the two semantics are therefore equivalent. However $\mathcal{O}(P)$ contains more useful information. On one side, it correctly models computed answers. On the other side, it has nice properties also from the model-theoretic viewpoint. This can be shown by considering the properties of the (atomic logical consequences) semantics $\mathcal{O}_2$ and the relation between $\mathcal{O}$ and $\mathcal{O}_2$.

Theorem 3.25 *[47, 62] Let P be a positive program and A be a (possibly non-ground) atom. Then $P \models \forall A$ iff $A \in \mathcal{O}_2(P)$.*

Theorem 3.26 *[47] Let P be a positive program. Then*
$\mathcal{O}_2(P) = \{A | \exists B \in \mathcal{O}(P)$ and $\exists \vartheta$ such that $A = B\vartheta\}$

This allows us to determine from $\mathcal{O}(P)$ the *correct answer substitutions*, as shown by the following corollary, which can easily be derived from theorems 3.25 and 3.26.

Corollary 3.27 *Let P be a program and $G = A_1, \ldots, A_n$ be a goal. Then ϑ is a correct answer substitution for G in P (i.e. $P \models \forall(A_1 \wedge \ldots \wedge A_n)\vartheta$) iff all the atoms $A_i\vartheta$ are instances of atoms in $\mathcal{O}(P)$.*

Note that, as shown in [88], correct answer substitutions cannot be determined from the least Herbrand model.

4 A compositional semantics

The semantics defined in section 3 is compositional w.r.t. the AND operator. We consider here $\cup$-compositionality, i.e. composition w.r.t. to union of programs. A semantics $\mathcal{S}$ is compositional w.r.t. the union of programs, if for any pair of programs P_1 and P_2, $\mathcal{S}(P_1 \cup P_2)$ can be derived from $\mathcal{S}(P_1)$ and $\mathcal{S}(P_2)$. The semantics $\mathcal{O}$ that we have considered so far is not compositional w.r.t. the union of programs, as shown by the following example.

Example 4.1 *Consider the following programs.*

$$P = \{ \quad r(b) : -p(b). \quad p(a). \quad \} \qquad Q = \{ \quad p(b). \quad \} \qquad R = \{ \quad p(a). \quad \}$$

$\mathcal{O}(P) = \mathcal{O}(R) = \{p(a)\}$
$\mathcal{O}(Q) = \{p(b)\}$
$\mathcal{O}(P \cup Q) = \{p(a), p(b), r(b)\}$ *while* $\mathcal{O}(R \cup Q) = \{p(a), p(b)\}$. *The same problem arises if we consider* $\mathcal{O}_1$ *or* $\mathcal{O}_2$.

Example 4.1 shows that $\mathcal{O}$ does not contain enough information to be able to model $\cup$-composition. This can formally be shown by considering the $\cup$-compositional observational equivalence $\simeq_\cup$, given in definition 4.2 for the computed answers observable, and by proving that $\mathcal{O}$ is not correct w.r.t. $\simeq_\cup$.

Definition 4.2 *Let* P_1, P_2 *be positive programs.* $P_1 \simeq_\cup P_2$ *if, for every positive goal* G *and for every program* Q, $G \overset{\vartheta}{\longmapsto}_{P_1 \cup Q} \square$ *iff* $G \overset{\vartheta'}{\longmapsto}_{P_2 \cup Q} \square$ *and* $\vartheta = (\vartheta'\rho)_{|G}$, *where* ρ *is a renaming.*

From definition 4.2 one can note that a semantics correct w.r.t. $\simeq_\cup$ is essentially a function from interpretations to interpretations. As a matter of fact two $\cup$-compositional semantics (correct w.r.t. the successful derivations observable) are the semantics in which the denotation of P is the associated immediate consequences operator T_P and the functional semantics defined in [81]. Gaifman and Shapiro first suggested to use sets of (equivalence classes of) clauses as a representation of one such a function, modeling the successful derivations [62] and the computed answers [63] observables. This idea fits quite naturally within the s-semantics approach since the semantic domains are syntactic objects, i.e. programs.

The Ω-*semantics* [22, 21] is similar to one of the semantics in [62], yet it is defined according to the general s-semantics approach. It was originally defined for a more general composition operator $\cup_\Omega$, defined on Ω-open programs. An Ω-*open program* [22] P is a positive program in which the predicate symbols belonging to the set Ω are considered partially defined in P. P can be composed with another program Q which may further specify the predicates in Ω and use clauses in P to complete its own predicate definitions. Such a composition is denoted by $\cup_\Omega$. Formally, given the Ω-open programs P_1, P_2, if $Pred(P_1) \cap Pred(P_2) \subseteq \Omega$ then $P_1 \cup_\Omega P_2$ is the Ω-open program $P_1 \cup P_2$, otherwise $P_1 \cup_\Omega P_2$ is undefined. A more general notion of composition which allows different sets of open predicates for the composed programs is considered in [21]. The semantics of open programs must be compositional w.r.t. $\cup_\Omega$, i.e. the semantics of $P_1 \cup_\Omega P_2$ must be derivable from the semantics of P_1 and P_2. Note that if Ω contains all the predicate symbols then $\cup_\Omega$ is the same as the standard union.

The $\cup_\Omega$-compositional observational equivalence $\simeq_\Omega$ of definition 4.3 is the straightforward extension of definition 4.2.

Definition 4.3 *Let P_1, P_2 be Ω-open programs. $P_1 \simeq_\Omega P_2$ if, for every positive goal G and for every program Q such that, for $i = 1, 2$, $P_i \cup_\Omega Q$ is defined, $G \stackrel{\vartheta}{\longmapsto}_{P_1 \cup_\Omega Q} \Box$ iff $G \stackrel{\vartheta'}{\longmapsto}_{P_2 \cup_\Omega Q} \Box$ and $\vartheta = (\vartheta'\rho)_{|G}$, where ρ is a renaming.*

The above observational equivalence is captured by the following operational semantics. We denote by Id_Ω the set of clauses $\{p(\tilde{X}) : -p(\tilde{X}) \mid p \in \Omega\}$ where Ω is a set of predicate symbols.

Definition 4.4 (Ω-compositional computed answer substitutions semantics) *[21] Let P be a positive program, Ω be a set of predicate symbols, P^* be the augmented program $P \cup Id_\Omega$ and R be a fair selection rule. Then we define*

$$\begin{aligned}\mathcal{O}_\Omega(P) = \{c \mid\ & \exists \tilde{X} \in V, \\ & \exists \text{ a derivation} \\ & p(\tilde{X}) \stackrel{\gamma}{\leadsto}_{P,R} D_1, \ldots, D_m \stackrel{\vartheta}{\leadsto}_{P^*,R} B_1, \ldots, B_n, \\ & \text{and } Pred(B_1, \ldots, B_n) \subseteq \Omega, \\ & c = p(\tilde{X})\gamma\vartheta : -B_1, \ldots, B_n\}\end{aligned}$$

Note that $\mathcal{O}_\Omega(P)$ is a set of *resultants* [89, 3] obtained from goals of the form $p(\tilde{X})$ in P and is strongly related to partial evaluation [78][3].

The set of clauses Id_Ω in the previous definition is used to delay the evaluation of open atoms. This is a trick which allows us to obtain a denotation which is independent from the (fair) selection rule.

The semantic domain $\mathcal{C}_\Omega$ for the denotation $\mathcal{O}_\Omega(P)$ is the set of clauses whose body predicates are all in Ω (*conditional atoms*) modulo the following equivalence $\approx_+$.

Definition 4.5 *Assume $c_1 = A_1 : -B_1, \ldots, B_n$ and $c_2 = A_2 : -D_1, \ldots, D_n$. Then $c_1 \approx_+ c_2$ iff $\exists$ a renaming ρ such that $A_1 = A_2\rho$ and $\{\!|B_1, \ldots, B_n|\!\} = \{\!|D_1\rho, \ldots, D_n\rho|\!\}$ where $\{\!|\ |\!\}$ denotes a multiset.*

Definition 4.6 *A π-interpretation for an Ω-open program P is any subset of $\mathcal{C}_\Omega$.*

$\mathcal{O}_\Omega(P)$ is then a π-interpretation for Ω-open programs. Note that we consider bodies of clauses as multisets.

Example 4.7 *Consider the following Ω-open program P, where $\Omega = \{q\}$.*

$$\begin{aligned}P = \{\ & p(X) : -q(X). \\ & r(X) : -s(X). \\ & q(a). \\ & s(b). \qquad\qquad \}\end{aligned}$$

Then $\mathcal{O}_\Omega(P) = \{p(X) : -q(X), \quad p(a), \quad q(a), \quad r(b), \quad s(b)\}$.

The following results show that $\mathcal{O}_\Omega$ actually models computed answer substitutions in a compositional way.

Theorem 4.8 (compositionality) *[21] Let P, P_1, P_2 be programs and assume $Pred(P_1) \cap Pred(P_2) \subseteq \Omega$. Then the following facts hold*

- $\mathcal{O}_\Omega(\mathcal{O}_\Omega(P_1) \cup_\Omega \mathcal{O}_\Omega(P_2)) = \mathcal{O}_\Omega(P_1 \cup_\Omega P_2)$,

[3] The relation between the semantics and partial evaluation will be discussed in section 5.3.

- $P \simeq_\Omega \mathcal{O}_\Omega(P)$.

As usual $\mathcal{O}_\Omega(P)$ can be characterized as the least fixpoint of an immediate consequences operator. We can simply define such an operator in terms of the unfolding rule of definition 3.16. Note that we consider a π-interpretation also as a set of (renamed apart) syntactic clauses. Moreover operators such as unf_P are considerd as operators on $\mathcal{C}_\Omega$. These "semantic" versions are well defined since clauses are always renamed apart.

Definition 4.9 *[21] Let P be an Ω-open program and let $I \subseteq \mathcal{C}_\Omega$. Then*

$$T^\pi_{P,\Omega} = unf_P(I \cup Id_\Omega).$$

Lemma 4.10 *[21] Let P be an Ω-open program. Then $T^\pi_{P,\Omega}$ is continuous on $(\Im, \subseteq)$).*

Definition 4.11 *[22, 21] The (least) fixpoint semantics of an Ω-open program P is defined as $\mathcal{F}(P) = T^\pi_{P,\Omega} \uparrow \omega$.*

Theorem 4.12 (equivalence of the fixpoint and the operational semantics) *[21] Let P be an Ω-open program. Then $\mathcal{F}(P) = \mathcal{O}_\Omega(P)$.*

The denotation $\mathcal{O}_\Omega(P)$ can be viewed as a function which, when provided with the denotation of a program Q, returns the denotation of $P \cup_\Omega Q$. If we move from denotations to Herbrand models, we can associate to the denotation (π-interpretation) I the set of the least Herbrand models of all the programs which can be obtained by "completing" the denotation I (considered as a program), by taking the union of I with a suitable set of ground atoms defining the open predicates. This is formalized by the function $\mathcal{H}$ in the following definition.

Definition 4.13 *[21] Let I be a π-interpretation for an Ω-open program. Then $\mathcal{H}(I) = \{\mathcal{O}_1(I \cup_\Omega J)\}$ where J is any set of ground atoms $p(\tilde{t})$ such that $p \in \Omega$ and $p(\tilde{t})$ is an instance of an atom in the body of a clause in I.*

If we consider the program P of example 4.7 on the signature S, defined by $C = \{a\backslash 0, b\backslash 0\}$, then
$\mathcal{H}(\mathcal{O}_\Omega(P))= \{\{p(a), q(a), r(b), s(b)\}, \{p(a), q(a), p(b), r(b), s(b), q(b)\}\}$.

π-models are then those defined according to definition 2.1 and have the following properties.

Proposition 4.14 *[21] Let P be an Ω-open program. The following statements hold*

- *every Herbrand model of P is a π-model of P,*
- *$\mathcal{O}_\Omega(P)$ is a π-model of P.*

The main idea behind the compositional semantics is the use of sets of clauses as semantic domain. This is the syntactic device which allows us to obtain a unique representation for a possibly infinite set of Herbrand models when a unique representative Herbrand model does not exist. Similar domains consisting of clauses have been used to model non-standard observables [61, 59] (see section 5.3) and to characterize logic programs with negation [43, 75, 60] (with the aim of delaying the evaluation of negative literals).

The delayed evaluation of open predicates which is typical of $\mathcal{O}_\Omega(P)$ can easily be generalized to other logic languages, to achieve compositionality w.r.t the union of programs. By modifying $\mathcal{O}_\Omega(P)$ we can obtain semantics compositional w.r.t. other composition operators, as for example

inheritance mechanisms [18] (see section 6.6). $\mathcal{O}_\Omega(P)$ can be considered as the semantic basis for modular program analysis, since by using suitable abstractions of $\mathcal{O}_\Omega(P)$, we can analyze program components and then combine the results to obtain the analysis of the whole program [32].

Let us finally mention that $\mathcal{O}_\Omega$ is strongly related to *abduction* [44]. If Ω is the set of abducible predicates, the abductive consequences of any goal G can be found by executing G in $\mathcal{O}_\Omega(P)$.

5 Other observables

5.1 Finite failures

There exist other useful observables for positive logic programs, such as, for example, *finite failures*. Indeed the standard semantics of positive logic programs should correctly model both the successful computed answers and the finite failures. The following definition formalizes the observational equivalence $\simeq_{FF}$ based on finite failures.

Definition 5.1 *Let P_1, P_2 be positive programs, G be a positive goal and $\mathcal{T}_1$ and $\mathcal{T}_2$ be SLD-trees (defined by a fair selection rule) for G in P_1 and P_2 respectively. Then $P_1 \simeq_{FF} P_2$ if for every goal G, $\mathcal{T}_1$ is finitely failed if and only if $\mathcal{T}_2$ is finitely failed.*

We will not consider the finite failure semantics, even because a correct and fully abstract generalization of the s-semantics modeling finite failures does not yet exist. Let us just mention that the (ground) *finite failure set* is not correct w.r.t. $\simeq_{FF}$, as shown by the following example.

Example 5.2 *Consider the following programs P_1 and P_2.*

$P_1 = \{\ p(f(X)) : -p(X).\quad q(a).\ \}$ $\qquad$ $P_2 = \{\ p(f(X)) : -p(X), p(a).\quad q(a).\ \}$

The Finite Failure set of both P1 and P2 is

$$\{p(a), p(f(a)), q(f(a)), p(f(f(a))), \ldots\},$$

while $P_1 \simeq_{FF} P_2$ does not hold, since the goal $p(X)$ finitely fails in P_2 only.

It can be shown that the *non-ground finite failure set* as defined in [84] is indeed correct w.r.t. $\simeq_{FF}$. However, the AND-compositionality property does not hold, i.e. it is not possible to decide whether a conjunctive goal finitely fails by just looking at the non-ground finite failure set. We believe that a correct and AND-compositional semantics for finite failure needs to be based on a semantics similar to the one of section 5.3.

5.2 Multisets of answers

The s-semantics was extended in [92] to deal with *multisets* rather than sets. Such an extension was needed to investigate properties which make possible improvements in the performance of the bottom-up fixpoint evaluation. Algorithms such as the Seminaive evaluation [9] try to avoid repeating inferences by comparing the new facts computed at each iteration with previously generated facts to eliminate duplicates. To study properties of these algorithms and their specializations for certain classes of programs, it is then necessary to consider duplicates, and hence multisets of atoms.

We show here the definition of the multiset version of the s-semantics (ms-semantics for short) from [92]. For the sake of uniformity we use a T_P-like construction. A more general

formulation which allows us to express different evaluation algorithms and different semantics is given in [92].

The *ms*-semantics can be obtained by simply replacing sets by multisets in all the definitions of section 3. Therefore, in the following an interpretation will be a multiset of atoms modulo variance and a program will be a multiset of clauses. We use $\{\!|\ |\!\}$ as multiset constructor, while $set(X)$ denotes the set obtained from the multiset X by ignoring multiplicities. In this section, $\in$ is used for multiset membership. For example $\{\!|n^2 \mid n \in \{\!|2,2,3|\!\}\ |\!\} = \{\!|4,4,9|\!\}$. Given an infinite chain $S_1 \subseteq S_2 \ldots$ of multisets, where $X \subseteq Y$ denotes multiset inclusion, its limit $S = lim_{n\to\infty} S_i$ is defined as the multiset S where the multiplicity of any $s \in S$ is the least upper bound (in $\mathcal{Z} \cup \{\infty\}$) of the multiplicities of s in S_n. In the following definition, as usual, we assume that all the atoms and all the clauses are renamed apart.

We denote by $mset(X)$ the set obtained from the multiset X by replacing any element a with multiplicity n by n (different) elements $a^1, \ldots, a^n$. When atoms are unified the superscripts are simply ignored.

Definition 5.3 *[92] Let P be a positive program and I be an interpretation. Then we define*

$$T_P^m(I) = \{\!|\ A \in \mathcal{B} \mid \begin{array}{l} H : -B_1, \ldots, B_n \text{ is a clause in } P, \\ \{C_1^{j_1}, \ldots, C_n^{j_n}\} \subseteq mset(I), \\ \vartheta = mgu((B_1, \ldots, B_n), (C_1, \ldots, C_n)) \text{ and } A = H\vartheta \end{array} |\!\}$$

Example 5.4 *Let P be the program*

$$P = \{\ p(a) : -q(a), q(a).\ \}$$

and I be the interpretation $I = \{\!|\ q(a),\ q(a)\ |\!\}$. Then

$$T_P^m(I) = \{\!|\ p(a),\ p(a),\ p(a),\ p(a)\ |\!\}.$$

The *ms*-semantics is defined as follows.

Definition 5.5 *Let P be a positive program. Then we define*

$$\mathcal{F}_m(P) = lim_{n\to\infty}\ T_P^m \uparrow n$$

By considering a suitable notion of complete lattice of multiset interpretations, the previous definition can be shown to correspond to the least fixpoint of T_P^m.

The *ms*-semantics $\mathcal{F}_m(P)$ contains all the (possibly repeated) computed answers for atomic goals of the form $p(\tilde{X})$. Repeated answers correspond to different "parallel" derivations which give the same computed answers for a given goal (by parallel derivation we mean a derivation where all the atoms in each resolvent are rewritten at each step).

Example 5.6 *Let P be the program*

$$P = \{\ \begin{array}{l} p(X) : -q(X), q(X). \\ p(a). \\ q(a). \end{array} \}$$

The ms-semantics of P is

$$\mathcal{F}_m(P) = \{\!|\ p(a),\ p(a),\ q(a)\ |\!\}$$

Accordingly, by using a parallel derivation, we can obtain the answer X/a for the goal $p(X)$ in the program P in two different ways (by using either the first or the second clause). Analogously, for the goal $p(X)$ in the program Q

$$Q = \{ \quad \begin{array}{l} p(X) : -p(X). \\ p(a). \end{array} \quad \}$$

we have infinitely many different ways to obtain the answer X/a (corresponding to derivations of increasing length). Then, the ms-semantics of Q is the infinite multiset

$$\mathcal{F}_m(Q) = \{\!|\ p(a),\ p(a), \ldots\ |\!\}$$

(while the s-semantics contains only one copy of $p(a)$).

By using a parallel derivation rule, we can then define an operational semantics equivalent to $\mathcal{F}_m(P)$ and hence an observational equivalence based on the "multiple answers" observable, for which the semantics $\mathcal{F}_m(P)$ would be fully abstract. Finally note that, as shown by the following proposition, the s-semantics can be obtained from the ms-semantics by ignoring multiplicities.

Proposition 5.7 *[92] Let P be a positive program. Then $\mathcal{F}(P) = set(\mathcal{F}_m(P))$.*

5.3 Resultants

We will consider now less abstract observables which make visible internal computation details. If we are only concerned with the input-output behavior of programs we should just observe computed answers and finite failures. However there are tasks, such as program analysis and optimization, where we are forced to observe and take into account other features of the derivation. In principle one could be interested in the complete information about the SLD-derivation, namely the sequences of goals, most general unifiers and variants of clauses. The *resultants*, introduced in [89] in the framework of partial evaluation, are a compact representation of the relation between the initial goal and the current $\langle goal, mgu \rangle$ pair. They are useful (see [3]) to formalize the properties of SLD-resolution. Our basic observable, for given goal G and selection rule R, will then be the set of all the pairs $\langle \mathcal{R}_i, \Sigma_i \rangle$, where $\mathcal{R}_i$ is a resultant derived from G by R and Σ_i is the corresponding sequence of clauses. We will then consider a semantics $\mathcal{O}^{\mathcal{R}}_R(P)$, defined according to the s-semantics approach, modeling the resultants. We obtain a kind of "collecting semantics" which gives the maximum amount of information on computations and allows us to observe all the internal details of SLD-derivations. It is essentially the collecting semantics with selection rule defined in [61, 59] extended with the information on the sequence of clauses.

As we will discuss later, several semantics useful for program analysis can be obtained by abstraction from $\mathcal{O}^{\mathcal{R}}_R(P)$. Let us first give the definition of resultant.

Definition 5.8 (Resultant with clauses) *Let P be a positive program, $G_1, \ldots, G_n$ be a goal and R be a selection rule. If there exists an SLD-derivation (using the rule R) of the goal $B_1, \ldots, B_m,\ m \geq 0$ from $G_1, \ldots, G_n$ and if the derivation computes the answer ϑ and is obtained by using the sequence of clauses $[c_1, \ldots, c_k], k \geq 0$ (denoted by $G_1, \ldots, G_n \overset{\vartheta}{\leadsto}{}^{[c_1,\ldots,c_k]}_{P,R}$ $B_1, \ldots, B_m,\ m, k \geq 0$), then $\langle (G_1 \wedge \ldots \wedge G_n)\vartheta \leftarrow B_1, \ldots, B_m, [c_1, \ldots, c_k] \rangle$ is a resultant with clauses of the goal $G_1, \ldots, G_n$ in the program P with selection rule R.*

Note that we denote by $G_1, \ldots, G_n \overset{\varepsilon}{\leadsto}{}^{[\,]}_{P,R} G_1, \ldots, G_n$ a derivation of length 0 and hence we consider also the resultants with clauses of the form
$\langle G_1 \wedge \ldots \wedge G_n \leftarrow G_1, \ldots, G_n, [\,] \rangle$.

The set of resultants is clearly dependent upon the selection rule. If we take the selection rule into account, the ordering of atoms in the goal (and in the body of a clause) is relevant. Therefore, the right hand sides of resultants are sequences of atoms. Note that the resultant is a definite clause (with the body viewed as a sequence of atoms) if the initial goal is atomic. The observable for a goal G in a program P with a selection rule R is the set $\mathcal{R}^G_{(R,P)}$ of all the resultants with clauses for G in P via R. Resultants which are variants of each other are equivalent.

We can now define the observational equivalence.

Definition 5.9 *Let P_1, P_2 be positive programs and R be a selection rule. Then $P_1 \simeq_{\mathcal{R}} P_2$ if for every goal G, $\mathcal{R}^G_{(R,P_1)} = \mathcal{R}^G_{(R,P_2)}$.*

In order to obtain the top-down definition of a semantics $\mathcal{O}^{\mathcal{R}}_R(P)$ correct w.r.t. $\simeq_{\mathcal{R}}$, we use the s-semantics technique, namely we consider the sets of resultants with clauses for atomic goals of the form $p(\tilde{X})$. We will show later that this denotation allows us to determine the observable for any goal. The semantic domain $\mathcal{C}$ is then the set of all the (equivalence classes of) pairs composed of a clause and a sequence of clause identifiers and a π-interpretation is any subset of $\mathcal{C}$.

Definition 5.10 *Let P be a positive program and R be a selection rule. Then*

$$\mathcal{O}^{\mathcal{R}}_R(P) = \{ \langle \mathcal{R}, \Sigma \rangle \mid p(\tilde{X}) \overset{\vartheta}{\leadsto}{}^{[c_1,\ldots,c_k]}_{P,R} B_1, \ldots, B_m,\ m, k \geq 0,\ \mathcal{R} = p(\tilde{X})\vartheta : -B_1, \ldots, B_m,\ \Sigma = [c_1, \ldots, c_k] \}$$

Consider the program in the following example.

Example 5.11 $P = \{ c_1 = p(a). \quad c_2 = p(X) : -r(X), q(X,Y). \quad c_3 = q(b,a). \quad c_4 = r(b). \}$

If we choose the leftmost selection rule $\leftarrow$, definition 5.10 gives the following denotation.

$$\begin{aligned}\mathcal{O}^{\mathcal{R}}_{\leftarrow}(P) = \{ & \langle p(X) : -p(X), [] \rangle, \quad \langle p(a), [c_1] \rangle, \\ & \langle p(X) : -r(X), q(X,Y), [c_2] \rangle, \\ & \langle p(b) : -q(b,Y), [c_2, c_4] \rangle, \quad \langle p(b), [c_2, c_4, c_3] \rangle, \\ & \langle q(X,Y) : -q(X,Y), [] \rangle, \quad \langle q(b,a), [c_3] \rangle, \\ & \langle r(X) : -r(X), [] \rangle, \quad \langle r(b), [c_4] \rangle \}\end{aligned}$$

$\mathcal{O}^{\mathcal{R}}_R(P)$ can be proved to be correct w.r.t. $\simeq_{\mathcal{R}}$. As a matter of fact, since $\mathcal{O}^{\mathcal{R}}_R(P)$ is essentially the collecting semantics with selection rule defined in [61, 59], all the theorems proved in [59] can easily be extended to our definition. In particular, if we want a bottom-up definition equivalent to the top-down one, we have to consider "local" selection rules only. A local selection rule is defined in [112] as a rule which always selects in a goal N one of the most recently introduced atoms in the derivation from the initial goal to N. Note that the PROLOG leftmost rule is local and that in general local rules produce SLD-trees with a simpler structure, suitable for efficient searching techniques [112]. For the sake of simplicity, we will give the next definitions in the case of the leftmost selection rule only. The general complete formalization can be found in [59].

The intuition behind the immediate consequences operator in definition 5.12 is the following. We can unfold the atom B_k in the clause $H : -B_1, \ldots, B_k, \ldots, B_n$ if all the atoms B_j, $j = 1, \ldots, k-1$ have been (completely) evaluated and have therefore already unit clauses among their resultants. The resultants with clauses of level 0, for a program P with the set of predicate symbols Π, are given by the π-interpretation $Id = \{\langle p(\tilde{X}) : -p(\tilde{X}), [] \rangle \mid p \in \Pi\}$. In definition 5.12 both clauses and resultants from $X \cup Id$ are standardized apart.

Definition 5.12 *Let P be a positive program and $X \subseteq \mathcal{C}$. Then*

$$T_{(P,\mathcal{R})}(X) = Id \bigcup \{\langle \mathcal{R}, \Sigma \rangle \mid \begin{array}{l} \exists c \ = \ A : -B_1, \ldots, B_k, \ldots, B_m \in P, \\ \exists \ \langle B'_1, \Sigma_1 \rangle, \ldots, \langle B'_{k-1}, \Sigma_{k-1} \rangle \in X, \\ \exists \ \langle B'_k : -D_1, \ldots, D_n, \Sigma_k \rangle \in X \cup Id, \\ \vartheta = mgu((B_1, \ldots, B_k), (B'_1, \ldots, B'_k)), \\ \mathcal{R} \ = \ (A : -D_1, \ldots, D_n, B_{k+1}, \ldots, B_m)\vartheta, \\ \Sigma \ = \ [c] :: \Sigma_1 :: \ldots :: \Sigma_k \qquad \} \\ \textit{(where :: denotes concatenation of sequences)} \end{array}$$

Since the operator $T_{(P,\mathcal{R})}$ is continuous on the lattice of π-interpretations, we can define the fixpoint semantics of P, $\mathcal{F}_{\mathcal{R}}(P)$, as the least fixpoint of $T_{(P,\mathcal{R})}$ in the usual way. The following theorem shows the equivalence of the top-down and bottom-up semantics, while theorem 5.14 shows that the denotation $\mathcal{F}_{\mathcal{R}}(P)$ actually collects all the information on the resultants in SLD-derivations using the leftmost selection rule. The proofs of both theorems can easily be obtained from the proofs of Theorems 23 and Lemma 22 in [61].

Theorem 5.13 *Let P be positive program. Then $\mathcal{O}^{\mathcal{R}}_{\leftarrow}(P) = \mathcal{F}_{\mathcal{R}}(P)$.*

Theorem 5.14 *Let P be a positive program and $G = A_1, \ldots, A_m$ be a goal. Then $\langle \mathcal{R}, \Sigma \rangle$ is a resultant with clauses of goal G in P via the leftmost selection rule iff $\exists \{\langle H_1, \Sigma_1 \rangle, \ldots, \langle H_{s-1}, \Sigma_{s-1} \rangle, \langle H_s : -B_1, \ldots, B_k, \Sigma_s \rangle\} \in \mathcal{F}_{\mathcal{R}}(P)$*
such that
$\vartheta = mgu((A_1, \ldots, A_s), (H_1, \ldots, H_s))$,
$\mathcal{R}' = ((A_1 \wedge \ldots \wedge A_m) \leftarrow B_1, \ldots, B_k, A_{s+1}, \ldots, A_m)\vartheta$,
$\Sigma' = \Sigma_1 :: \ldots :: \Sigma_s$,
$\Sigma = \Sigma'$ *and* $\mathcal{R}$ *is a variant of* $\mathcal{R}'$.

Let us finally mention that, from the model theory point of view, one can define the following function from π-interpretations to Herbrand interpretations.

Definition 5.15 *Let I be a π-interpretation. Then $\mathcal{H}(I)$ is the set consisting of the set of ground instances of the unit resultants in I.*

By using the notion of π-model given by definition 2.1, we have the following result, which shows that the semantics modeling different observables are all π-models, yet provide different information on the observable program behavior.

Proposition 5.16 *Let P be a program. Then $\mathcal{O}_1(P)$ (the least Herbrand model of P), $\mathcal{O}(P)$ (the computed answers semantics of P) and $\mathcal{O}^{\mathcal{R}}_R(P)$ (the resultants semantics of P) are all π-models of P.*

As already mentioned, both the resultants semantics and the compositional semantics of section 4 are strongly related to *partial evaluation*, a program transformation technique first

applied to logic programs in [78] and later fully formalized in [89]. The result of partial evaluation is a (finite) set of resultants, obtained from a program P and an atomic goal A. The selected set of resultants corresponds to a "cut" of the SLD-tree. A is atomic but not necessarily of the form $p(\tilde{X})$. The aim of partial evaluation is in fact to obtain a specialization of P for the goal A. The construction of the compositional semantics of section 4 and of the resultants semantics is based on goals of the form $p(\tilde{X})$ which trivially satisfy the *A-closedness* condition [89], which guarantees the completeness of partial evaluation. The relation between the procedural behaviors of a program and of its (compositional and resultants) semantics can then be understood in terms of soundness and completeness of partial evaluation.

5.3.1 Finite success

Let us give now an example of an observable semantics which can be derived as an abstraction of $\mathcal{O}_R^{\mathcal{R}}$. If we want to characterize *finite success* [40] we must be able to distinguish between unit resultants (representing successful derivations) and non-unit resultants (representing possibly non-terminating computations). Non-atomic resultants are abstracted upon resorting to the notion of *hypothetical* atoms. Each resultant of the form $A\texttt{:-}\tilde{B}$ is represented as the hypothetical atom $?A$. $?A$ conveys all the relevant information provided by $A\texttt{:-}\tilde{B}$ (that the associated derivation is partial) and abstracts from the body $\tilde{B}$, which is in fact irrelevant in this context. The extended Herbrand base $\mathcal{B}_E$ consists of hypothetical as well as standard atoms.

Interpretations are defined as subsets of the extended base $\mathcal{B}_E = \mathcal{B} \cup ?\mathcal{B}$, where $?\mathcal{B} = \{?A \mid A \in \mathcal{B}\}$. Two selectors, $Certain$ and $Uncertain$ are used to project any subset I of $\mathcal{B}_E$ into one of the base components.

$$Certain(I) = \{A \mid A \in \mathcal{B} \cap I\} \quad Uncertain(I) = \{A \mid A \in ?\mathcal{B} \cap I\}.$$

The *frontier semantics* $\mathcal{E}$ defined in [40] is obtained by collecting information computed at each iteration of the immediate consequences operator. Let F_i be the abstraction of the frontier computed at the i-th iteration level, then

$$\mathcal{E} = \bigcup_{i=0,\ldots} C_i \ \cup \ \bigcap_{i=0,\ldots} U_i$$

where $C_i = Certain(F_i)$ and U_i is the set of all the hypothetical atoms which unify with elements of $Uncertain(F_i)$. Thus $Certain(\mathcal{E})$ is the s-semantics while $Uncertain(\mathcal{E})$ contains all the atomic goals whose SLD-tree has at least one infinite branch. Clearly $\mathcal{E}$ captures finite success and failure of both ground and non ground atoms.

Theorem 5.17 *[40] Let P be a positive program and A be a non-ground atom.*

- *A unifies with $A_1, \ldots, A_n$ in $\mathcal{E}$ with mgu $\theta_1, \ldots, \theta_n$ respectively, and $?A \notin \mathcal{E}$ iff the goal A has an SLD-tree of finite success with c.a.s. $\theta_1, \ldots, \theta_n$.*
- *A unifies with $A_1, \ldots, A_n$ in $\mathcal{E}$ and $?A \in \mathcal{E}$ iff the goal A has a successful SLD-tree with at least one infinite branch.*
- *A does not unify with any atom in $\mathcal{E}$ and $?A \notin \mathcal{E}$ iff the goal A has a finitely failed SLD-tree.*
- *A does not unify with any atom in $\mathcal{E}$ and $?A \in \mathcal{E}$ iff the goal A has an SLD-tree with no success branches but at least an infinite one.*

Example 5.18 *Consider the program P consisting of the following clauses.*
$P = \{ \quad p(a)., \quad p(b) : -p(b)., \quad q(a). \quad \}$
$\mathcal{E} = \{ \quad p(a), \quad ?p(X), ?p(b), \quad q(a) \quad \}.$
We can note that $q(X)$ has finite success, $p(X)$ succeeds with an infinite branch, $q(b)$ finitely fails and $p(b)$ fails.

The construction of $\mathcal{E}$ recalls the theoretical characterization of termination of logic programs developed by Vasak e Potter in [110]. They compare terminating queries under different choices of the selection rule (thus dealing with different notions of universal termination) while we consider fair selection rules in theorem 5.17 and Prolog selection rule in section 6.5. Another difference lays on the fact that we use a single immediate consequences operator in the style of the s-semantics approach while they use various bottom-up constructions similar to the c-semantics (see Definition 3.6). Moreover, they do not obtain a specific goal independent denotation such as $\mathcal{E}$, which encompasses all the necessary information (as shown in theorem 5.17) to characterize success, finite and infinite failure.

5.3.2 Other abstractions of the resultants semantics

Several other existing equivalent top-down and bottom-up semantics can be derived as abstractions of $\mathcal{O}_R^{\mathcal{R}}$, including

- the *resultants semantics* defined (for any local rule R) in [61, 59], where we don't care about the sequences of clauses,
- the *resultants semantics with depth* defined (for the leftmost rule) in [11], where a sequence of clauses is abstracted by its length,
- the *partial answers semantics* $\mathcal{O}_R^{\mathcal{PA}}$ defined (for any local rule R) in [61, 59], where we only keep the heads of the resultants by labeling as partial those heads that were heads of a non-unit resultant,
- the *call patterns semantics* $\mathcal{O}_R^{\mathcal{CP}}$ defined in [61, 59], where (in the case of the leftmost selection rule) we delete all the atoms in the clause bodies but the first.

We list in the following some of the program properties which can be studied on the above semantics.

- The *call patterns*, i.e the procedure calls, for a goal G can be determined from $\mathcal{O}_R^{\mathcal{CP}}$. Let $H : -B_1$ be a clause in $\mathcal{O}_R^{\mathcal{CP}}$. Then if $\exists \vartheta = mgu(G, H)$, then $B_1\vartheta$ is a call pattern. The knowledge about the call patterns is useful in program optimization. The above property makes feasible a bottom-up characterization of (possibly abstract versions of) the call patterns.
- The *partial answers*, originally defined in [46], are the answers computed at any intermediate computation step. They can be determined from the partial answers semantics $\mathcal{O}_R^{\mathcal{PA}}$ as follows [61, 59]. ϑ is a partial answer for a goal $G_1, \ldots, G_n$ iff there exist $\{H_1, \ldots, H_n\} \in \mathcal{O}_R^{\mathcal{PA}}$ such that $\vartheta = mgu((G_1, \ldots, G_n), (H_1, \ldots, H_n))$. Partial answers are useful in program analysis and to characterize the semantics of concurrent languages.
- A goal G has the *universal termination* property iff there exists a frontier of a partial SLD-tree for G (obtainable using a suitable abstraction of the resultant semantics and theorem 5.14), such that all the atoms in the frontier are not labeled as partial answers. This

information is very important for the semantics of PROLOG [11, 17] and of all-solutions metapredicates [40].

- A goal G finitely fails iff
  - there exist a finite number of frontiers for G,
  - all the atoms in the frontiers of G are labeled as partial.

  This information is useful to get a bottom-up characterization of SLDNF-resolution [95]. The information in the frontiers can also be useful to get a fixpoint characterization of constructive negation.

6 Extending the s-semantics to other logic languages

6.1 Constraint logic programs

The s-semantics extends quite naturally to the *Constraint Logic Programming* paradigm as defined by Jaffar and Lassez [71], where constraints are interpreted over an algebraic structure $\mathcal{A}$. A constraint c is *solvable* iff there exists a valuation ϑ (*solution*) mapping variables to elements of the domain of $\mathcal{A}$, such that $c\vartheta$ is true in $\mathcal{A}$. We denote by $sol(c)$ the set of solutions of the constraint c. A CLP derivation step of a goal $c\Box A_1, \ldots, A_n$ in a program P results in a goal of the form $\tilde{c}\Box\tilde{B}_1, \ldots, \tilde{B}_n$, if there exist n (renamed apart) clauses in P, $H_i : -c_i\Box\tilde{B}_i, i = 1, \ldots, n$, such that $\tilde{c} = c \wedge c_1 \wedge \ldots \wedge c_n \wedge A_1 = H_1 \wedge \ldots \wedge A_n = H_n$ is solvable ($p(\tilde{t}) = p(\tilde{l})$ is an abbreviation for the unification atom $= (\tilde{t}, \tilde{l})$).

A successful *derivation* of a goal G (denoted by $G \stackrel{\Re}{\longmapsto}_P c\Box$) is a finite sequence of goals such that every goal is obtained from the previous one by means of a derivation step and the last goal has the form $c\Box$ where c is the *answer constraint*. The observable we consider is then the answer constraint. All the definitions and results on the answer constraint semantics are from [55]. The observational program equivalence $\simeq$ based on answer constraints is the following.

Definition 6.1 *Let P_1, P_2 be CLP programs. $P_1 \simeq P_2$ iff for every goal G the following hold*

- *if $G \stackrel{\Re}{\longmapsto}_{P_1} c\Box$ and $\vartheta \in sol(c)$ then $G \stackrel{\Re}{\longmapsto}_{P_2} c'\Box$ and there exists $\gamma \in sol(c')$ such that $\vartheta_{|Var(G)} = \gamma_{|Var(G)}$, and vice versa.*

Definition 6.2 (Answer constraint semantics) *Let P be a CLP program.*

$$\mathcal{O}(P) = \quad \{p(\tilde{X}) : -c \in \mathcal{B} \quad \mid true\Box p(\tilde{X}) \stackrel{\Re}{\longmapsto}_P c\Box \quad \}.$$

The interpretation base $\mathcal{B}$ is now the set of all the $\sim$ equivalence classes of *constrained atoms* (CLP unit clauses of the form $p(\tilde{X}) : -c$). A π-interpretation is any subset of $\mathcal{B}$. The equivalence $\sim$ is introduced in order to abstract from irrelevant syntactical details and is defined as $p(\tilde{X}) : -c_1 \sim p(\tilde{Y}) : -c_2$ iff for any solution ϑ of c_1 there exists a solution γ of c_2 such that $p(\tilde{X})\vartheta = p(\tilde{Y})\gamma$ and vice versa. Note that the previous definition of $\sim$ is semantic. The existence of a syntactic representation for $\sim$ depends on $\mathcal{A}$ (e.g. variance for the Herbrand universe). $\mathcal{O}$ is correct (and fully abstract) w.r.t. answer constraints. Note that this semantics was not considered in the original report on the CLP semantics [72]. The usual AND-compositionality holds for $\mathcal{O}$.

Theorem 6.3 *[55] Let P be a CLP program and $G = c_0 \Box A_1, \ldots, A_n$ be any goal. Then $G \overset{\Re}{\longmapsto}_P c_{ans} \Box$ iff there exist n (renamed apart) constrained atoms $B_i : -c_i \in \mathcal{O}(P)$, $i = 1, \ldots n$, such that for any $\vartheta \in sol(c_{ans})$ there exists $\gamma \in sol(c_0 \wedge c_1 \wedge \ldots \wedge c_n \wedge A_1 = B_1 \wedge \ldots \wedge A_n = B_n)$ such that $\vartheta_{|Var(G)} = \gamma_{|Var(G)}$, and vice versa.*

The immediate consequences operator of definition 6.4 allows us to define a fixpoint semantics equivalent to $\mathcal{O}$.

Definition 6.4 *Let P be a CLP program and I be a π-interpretation.*

$$
\begin{aligned}
T_P^{\pi}(I) = \{ \quad & p(\tilde{X}) : -c \in \mathcal{B} \mid \\
& \exists \text{ a renamed clause } p(\tilde{t}) : -c_0 \Box p_1(\tilde{t_1}), \ldots, p_n(\tilde{t_n}) \text{ in } P, \\
& \exists\, p_i(\tilde{X_i}) : -c_i \Box \in I, 1 \leq i \leq n \text{ which share no variables}, \\
& c = (c_0 \wedge \tilde{X_i} = \tilde{t_1} \wedge \ldots \wedge \tilde{X_n} = \tilde{t_n} \wedge c_1 \wedge \ldots \wedge c_n \wedge \tilde{X} = \tilde{t}) \\
& \text{is solvable} \qquad \}.
\end{aligned}
$$

The function $\mathcal{H}$, on which the model theory is based, maps $\mathcal{O}(P)$ onto the least $\mathcal{A}$-model of P. The following proposition holds.

Proposition 6.5 *Let P be a CLP program. Then every $\mathcal{A}$-model of P is a π-model of P. Moreover $\mathcal{O}(P)$ is a π-model of P.*

It is straightforward to extend also the compositional semantics. The equivalent top-down and bottom-up semantics modeling the answer constraints have also an elegant algebraic characterization oriented towards abstract interpretation [66], that will be discussed in section 7.2.

The s-semantics of CLP and its compositional version have been applied to obtain the semantics of two new instances of the CLP scheme, namely $CLP(\mathcal{H}/\mathcal{E})$ and $CLP(\mathcal{AD})$. $CLP(\mathcal{H}/\mathcal{E})$ [1, 2] is a logic + equational language, where constraints are equations to be solved in an equational theory and the constraint solver is a narrowing algorithm. $CLP(\mathcal{AD})$ [15] models a deductive database language with updates. The semantics provides a nice characterization of the intensional part w.r.t. the extensional one and of the notion of transaction. The corresponding equivalence notions can profitably be used to prove interesting properties of optimization procedures.

The approach has finally been applied to *concurrent constraint programs* as defined in [101], leading to the definition of equivalent top-down and bottom-up semantics, defined as sets of unit clauses [45, 57], which are trees of *ask* and *tell* constraints. The denotation correctly models computed answers, finite failures and deadlocks, even if it is not $\cup$-compositional and fully abstract and there is no model-theoretic semantics.

6.2 Disjunctive logic programs

Disjunctive logic programs [90], where clause heads are disjunctions of atoms, have in general more than one minimal Herbrand model. We can get a unique model characterization by capturing the disjunctive consequences as a set of positive disjunctive ground clauses[4] (π-interpretations, called states in [98]), defined over the disjunctive Herbrand base.

Definition 6.6 (disjunctive Herbrand base) *[98] Let P be a disjunctive program. The* disjunctive Herbrand base *of P, denoted by DHB_P, is the set of all positive disjunctive ground clauses which can be formed using distinct ground atoms from the Herbrand base of P, such that no two logically equivalent clauses are in the set.*

[4] A positive disjunctive clause is a disjunctive clause with an empty body.

Definition 6.7 (π-interpretation, state) *[98] Let P be a disjunctive program. A* state *for P is a subset of DHB_P.*

Definition 6.8 *[98] Let P be a disjunctive program and I be a state.*

$$T_P^d(I) = \{C \in DHB_P \mid \begin{array}{l} C' : -B_1, \ldots, B_n \\ \quad \textit{is a ground instance of a clause in } P, \\ \{B_1 \vee C_1, \ldots, B_n \vee C_n\} \subseteq I, \\ C'' = C' \vee C_1 \vee \ldots \vee C_n, \\ \quad \textit{where } C_i, \forall i, 1 \leq i \leq n, \textit{ can be empty}, \\ C \textit{ is the smallest factor of } C''\} \end{array}$$

Example 6.9 *Let P be the disjunctive program*

$$P = \{ \begin{array}{l} p(X) \vee q(f(X)) : -r(X). \\ t(X) : -q(X). \\ p(b) \vee q(b). \\ r(a) \vee s(a). \end{array} \}$$

and I be the state $I = \{p(b) \vee q(b), r(a) \vee s(a)\}$.
$T_P^d(I) = \{p(b) \vee q(b), r(a) \vee s(a), p(a) \vee q(f(a)) \vee s(a), p(b) \vee t(b)\}$.

Theorem 6.10 *[98] Let P be a disjunctive program. T_P^d is continuous on the complete lattice $\langle 2^{DHB_P}, \subseteq \rangle$.*

Definition 6.11 *[98] Let P be a disjunctive program. The fixpoint semantics of P is $\mathcal{F}(P) = T_P^d \uparrow \omega$.*

Example 6.12 *Let P be the disjunctive program*

$$P = \{ \begin{array}{l} p(X,Y) \vee p(Z,Y) : -r(X,Y,f(Z)), q(Y). \\ r(a,b,f(c)). \\ q(b). \end{array} \}.$$

$\mathcal{F}(P) = \{q(b), r(a,b,f(c)), p(a,b) \vee p(c,b)\}$.

A state clearly represents a set of Herbrand interpretations. This can be formalized by defining the function $\mathcal{H}$ from states to states.

Definition 6.13 *Let P be a disjunctive program and I be a state for P. Then $\mathcal{H}(I)$ is the set of minimal Herbrand models of I (viewed as a disjunctive program).*

The following theorem is a straightforward consequence of some theorems in [98] and shows that the fixpoint semantics is indeed a π-model (called *model state* in [98]).

Theorem 6.14 *Let P be a disjunctive program. Then $\mathcal{H}(\mathcal{F}(P))$ is the set of all the minimal Herbrand models of P.*

Example 6.15 *One can easily check that by applying the function $\mathcal{H}$ to the fixpoint semantics of example 6.12 we obtain the Herbrand interpretations*
$\{q(b), r(a,b,f(c)), p(a,b)\}$ and $\{q(b), r(a,b,f(c)), p(c,b)\}$ which are exactly the minimal Herbrand models of the program P in example 6.12.

Theorem 6.14 shows the essence of the construction. As was the case for the compositional semantics of section 4, we obtain a unique denotation which syntactically represents all the relevant models. A similar mechanism, related to normal programs, will be considered in the next section.

6.3 Normal logic programs

We consider here the *semantic kernel* defined in [75] as a first step in the transformation of normal logic programs into constraint logic programs. It is a fixpoint construction which generalizes to the non-ground case the fixpoint semantics first proposed in [43][5]. The idea of the semantic kernel construction is to evaluate all the positive atoms in the clause bodies by unfolding them until there are no more positive atoms left. The semantic kernel is then a (possibly infinite) program consisting of negative clauses only[6]. The result of the transformation can be viewed as a π-interpretation (called quasi-interpretation in [75]).

Definition 6.16 (quasi-interpretation) *[75] Let P be a normal program. A* quasi-interpretation *for P is a set of negative clauses over the alphabet of P modulo variance.*

The semantic kernel is the least fixpoint of the immediate consequences operator T_P^k which maps quasi-interpretations onto quasi-interpretations.

Definition 6.17 (immediate consequences operator) *[75] Let P be a normal program and I be a quasi-interpretation. Then we define*

$$\begin{aligned} T_P^k(I) = \quad \{ & A\vartheta : -(\neg B_1^1, \ldots, \neg B_{h_1}^1, \ldots, \neg B_1^n, \ldots, \neg B_{h_n}^n, \neg B_1, \ldots, \neg B_m)\vartheta) \mid \\ & \exists A : -A_1, \ldots, A_n, \neg B_1, \ldots, \neg B_m \in P \\ & \exists\ A_i' : -\neg B_1^i, \ldots, \neg B_{h_i}^i \in I, i = 1, \ldots, n, \\ & \quad s.t.\ \vartheta = mgu((A_1, \ldots, A_n), (A_1', \ldots, A_n')) \qquad \} \end{aligned}$$

Definition 6.18 (semantic kernel) *[75]* $\mathcal{F}^k(P) = T_P^k \uparrow \omega$.

The semantic kernel is just an intermediate step in the process of defining a semantics for normal programs. It can be viewed as a compact representation of a set of models of the normal program, as shown by the following theorem.

Theorem 6.19 *[75] Every model of the completion of $\mathcal{F}^k(P)$ is a model of the completion of P.*

It is also strongly related to the *stable model* semantics [64] of P, as shown by the following very important theorem.

Theorem 6.20 *[43] Every Herbrand model of the completion of $\mathcal{F}^k(P)$ is a stable model of P.*

As we will show in the next section the semantic kernel construction can be useful even in relation to constructive negation.

6.4 Constructive negation

The inference rule for negation which is the most adequate to be handled by the s-semantics approach is clearly *constructive negation* introduced in [27, 28], because it allows the negative literals to compute answers.

The first attempt to extend the s-semantics to negation is described in [108]. It is a bottom-up semantics for stratified normal programs which generalizes to the non-ground case the construction of [4]. The resulting denotation has several similarities with the s-semantics,

[5] The same construction was independently proposed in [25].

[6] A *negative clause* [75] is a normal clause of the form $A : -\neg B_1, \ldots, \neg B_n$.

namely the fixpoint characterization and the use of sets of clauses (with constraints) as π-interpretations. However, there is no explicit relation to an observational equivalence based on an existing operational semantics (even if the reference derivation rule is clearly Chan's constructive negation). As a matter of fact, as it is the case for most declarative semantics of negation, the semantics in [108] tries to model the abstract intended meaning of the program and can be viewed as the ideal semantics to be approximated by effective operational semantics.

Essentially the same semantics (in the case of stratified programs) is obtained by the two-steps fixpoint construction in [60]. According to the last semantics, at each step we obtain a unique denotation, where some program fragments (the non-positive and the non-stratified fragments, respectively) are left uninterpreted. The first step consists of the fixpoint construction of the semantic kernel described in section 6.3, while the second step interprets the stratified component according to constructive negation, essentially following the approach in [108]. As a result of this step, the negation in the stratified component has been completely evaluated (and replaced by constraints), while the non-stratified negation is still there in some clauses.

The above approaches have been overriden by [23], which considers constructive negation in constraint logic programs as defined in [105], for which there exists a very strong completeness result w.r.t. 3-valued models of the completion. π-interpretations are pairs of sets of (equivalence classes of) constrained atoms (similar to those used in the CLP semantics discussed in section 6.1). The two elements of the pair specify the positive and negative components of the π-interpretation. The function $\mathcal{H}$ now maps π-interpretations onto *partial $\mathcal{A}$-interpretations* [50, 51]. The denotation $\mathcal{O}^{CN}(P)$ of a normal CLP program P has two equivalent top-down and bottom-up characterizations and is correct w.r.t. the answer constraints observable. Finally, $\mathcal{H}(\mathcal{O}^{CN}(P))$ is Kunen's semantics [80], namely $\Phi_P \uparrow \omega$, where Φ_P is Fitting's map on partial $\mathcal{A}$-interpretations [50]. It is worth noting that a similar bottom-up characterization can be obtained by the non-ground extension of Φ_P defined in [105].

6.5 PROLOG

We first consider pure PROLOG programs, i.e. programs without cut, built-in's or negation. Only the leftmost ($\leftarrow$) selection rule and the PROLOG search strategy are taken into account. The resultants semantics $\mathcal{O}^{\mathcal{R}}_{\leftarrow}(P)$ defined in section 5.3 contains enough information to capture the computational behavior of such programs. In fact it embeds the PROLOG selection rule, while the sequence of clauses associated to each resultant identifies a specific path in the partial SLD-tree. These paths can be ordered according to the lexicographic ordering induced by the ordering on program clauses. Moreover theorem 5.14 shows us how to select from the semantics the set of all the resultants with clauses of a given goal. Therefore, the semantics encodes the ordered trees of resultants for any goal. Clearly, if we are interested in some specific observable, the semantics $\mathcal{O}^{\mathcal{R}}_{\leftarrow}(P)$ contains too much information and can usefully be abstracted.

One such abstraction is presented in [17]. It has been designed to capture the set of (PROLOG) computed answer substitutions (p.a.s.) as observable, i.e. the set of answers which can be reached by using PROLOG's control. The observational equivalence induced by p.a.s. is the following.

Definition 6.21 *Let P_1, P_2 be pure PROLOG programs. $P_1 \approx_{p.a.s} P_2$ if for any goal G, θ is a p.a.s for G in P_1 if and only if θ is a p.a.s for G in P_2.*

We can reconstruct the semantics presented in [17], by first mapping $\mathcal{O}^{\mathcal{R}}_{\leftarrow}(P)$ into an ordered set of sequences of resultants, such that the i-th sequence represents the frontiers of the partial SLD-trees of depth i for the most general goals.

Example 6.22 *Consider the program P consisting of the following (sequence of) clauses:* $p(b,X).\ ::\ p(X,Y):-r(Y).\ ::\ p(c,Y).\ ::\ r(a):-p(a,a).\ ::\ r(b):-q(b).$

The frontiers of the partial SLD-trees are:

$f_0 = p(X,Y):-p(X,Y)\ ::r(X):-r(X)\ ::q(X):-q(X)$
$f_1 = p(b,X)\ ::p(X,Y):-r(Y)\ ::p(c,Y)\ ::r(a):-p(a,a)\ ::r(b):-q(b)$
$f_2 = p(b,X)\ ::p(X,a):-p(a,a)\ ::p(X,b):-q(b)\ ::p(c,Y)\ ::r(a):-r(a)$
$f_3 = p(b,X)\ ::p(X,a):-r(a)\ ::p(c,Y)\ ::r(a):-p(a,a)$
$f_4 = p(b,X)\ ::p(X,a):-p(a,a)\ ::p(c,Y)\ ::r(a):-r(a)$
$\ldots$

They are defined modulo variance and modulo the ordering among resultants with different predicate symbols in the head.

We may apply to each frontier the same abstraction introduced in section 5.3.1 for the frontier semantics $\mathcal{E}$. Namely, each non-atomic resultant of the form $A:-\tilde{B}$ is represented as the hypothetical atom $?A$.[7] An abstraction function $\sharp$ maps any sequence of resultants (clauses) into the corresponding sequence of abstractions.

Example 6.23 *Consider the program P of example 6.22. The abstract frontiers are:*

$f_0^\sharp = ?p(X,Y)\ ::?r(X)\ ::?q(X)$
$f_1^\sharp = p(b,Y)\ ::?p(X,Y)\ ::\ p(c,Y)\ ::?r(a)\ ::?r(b)$
$f_2^\sharp = p(b,Y)\ ::?p(X,a)\ ::?p(X,b)\ ::\ p(c,Y)\ ::?r(a)$
$f_3^\sharp = p(b,Y)\ ::?p(X,a)\ ::\ p(c,Y)\ ::?r(a)$
$f_4^\sharp = p(b,Y)\ ::?p(X,a)\ ::\ p(c,Y)\ ::?r(a)$
$\ldots$

Note that in the previous example $f_3^\sharp = f_4^\sharp = \ldots$, i.e. there are finitely many different abstractions of frontiers even if there are infinitely many partial SLD-trees for the goals $p(X,Y)$ and $r(X)$. This is not always the case. Consider, for instance, the following example.

Example 6.24 *Consider the program Q:* $p(0).\ ::\ p(s(X)):-p(X).$
There are infinitely many abstractions of frontiers $\{f_0,\ldots,f_j,\ldots\}$, where for each j

$$f_j = p(0)\ ::\cdots::\ p(s^{j-1}(0))\ ::?p(s^j(X)).$$

Any abstract frontier encodes a partial, yet safe, information on the p.a.s. of any goal. The following examples are meant to illustrate this fact.

Example 6.25 *Consider the (abstract) frontier $f_3^\sharp$ of example 6.23 and the goal $p(X,b)$. Recall that the hypothetical atom $?p(X,a)$ represents a node which could have descendants in the SLD-tree of the most general goal $p(X,Y)$. The goal $p(X,b)$ unifies with the first element $p(b,Y)$ of $f_3^\sharp$. This implies that $\{X/b\}$ is the first p.a.s. for $p(X,b)$. It does not unify with $p(X,a)$, hence it will also not unify with any possible descendant of the resultant abstracted by $?p(X,a)$. Since it unifies with $p(c,Y)$, its second answer will be $\{X/c\}$. No more answers are possible, since there are no other atoms with predicate symbol p in the sequence. Therefore $f_3^\sharp$ gives us a complete information on the p.a.s. for the goal $p(X,b)$.*

Example 6.26 *Consider now the non-atomic goal $G = p(X,b), p(X,Y)$ and the same frontier $f_3^\sharp$ of example 6.23. We first consider the first atom and extract information on it. In this case*

[7] Actually in [17] hypothetical atoms are called *divergent* and denoted by $\widehat{A}$. Here we adopt the notation introduced in section 5.3.1.

we will find the two answers $\{X/b\} :: \{X/c\}$. *Then we consider the corresponding instances of the second atom, i.e.* $p(X,Y)\{X/b\}$ *and* $p(X,Y)\{X/c\}$.

Since the goal $p(b,Y)$ *unifies with* $p(b,Y)$, *the empty substitution* ε *will be its first answer. Since it unifies also with* $p(X,a)$, *then we cannot exclude that it may enter an infinite loop after producing the first answer. Thus, even if the goal* $p(c,Y)$ *unifies with* $p(c,Y)$, *i.e. it has a first answer* ε, *the only safe answer for the goal* G *is* $\{X/b\}$ *since we cannot safely say that the other answer* $\{X/c\}$ *will be reached when executing* G *under the PROLOG's control.*

The reachability function $\rho_\perp$ formalizes these ideas. Let $Subst^*$ be the set of finite sequences of substitutions and $Subst^*_\perp = Subst^* \cup Subst^* :: \{\perp\}$ be the set of extended sequences, i.e. finite sequences which may end with the special symbol $\perp$, used to represent possible divergence. A strict concatenation $\odot$ is defined on elements of $Subst^*_\perp$.

Definition 6.27 *Let* $s_1, s_2 \in Subst^*_\perp$. *The strict concatenation* $\odot : Subst^*_\perp \mapsto Subst^*_\perp$ *is defined as:*

$$\odot(s_1,s_2) = \begin{array}{ll} s_1 :: s_2 & \textit{if } s_1 \in Subst^* \\ s_1 & \textit{otherwise.} \end{array}$$

For any goal G and abstract frontier S, $\rho_\perp(G,\, S)$ will return the sequence of p.a.s. for G which can be recognized as reachable by looking at the partial SLD-trees abstracted by S. The following definition is an extension of the one given in [17] for atomic goals only.

Definition 6.28 *Let* S *be a sequence in* $\mathcal{B}^*_E$, G *be a goal,* $A, A' \in \mathcal{B}$, $A_E \in \mathcal{B}_E$. *The reachability function* $\rho_\perp : \mathcal{B}^* \times \mathcal{B}^*_E \mapsto Subst^*_\perp$ *is the function inductively defined as follows:*

- *If* $G = A$ *is an atomic goal and* $S = \lambda$, *then* $\rho_\perp(G,\, \lambda) = \lambda$.
- *If* $G = A$ *is an atomic goal and* $S = A_E :: S'$, *then*

$$\rho_\perp(G,\, A_E :: S') = \begin{array}{ll} \theta :: \rho_\perp(A,\, S') & \\ \textit{if } A_E = A' \textit{ and } \theta = mgu(A,A')_{|vars(A)} & \end{array}$$

$$\rho_\perp(G,\, A_E :: S') = \begin{array}{ll} \perp & \\ \textit{if } A_E = ?A' \textit{ and there exists an } mgu(A,A') & \end{array}$$

$$\rho_\perp(G,\, A_E :: S') = \begin{array}{ll} \rho_\perp(A,\, S') & \\ \textit{otherwise.} & \end{array}$$

- *If* $G = A, \tilde{B}$ *is non-atomic, then*

$$\rho_\perp(G,\, S) = \begin{array}{ll} \rho_\perp(\tilde{B}\theta_1,\, S) \odot \cdots \odot \rho_\perp(\tilde{B}\theta_k,\, S) & \\ \textit{if } \rho_\perp(A,\, S) = \theta_1 :: \cdots :: \theta_k & \end{array}$$

$$\rho_\perp(G,\, S) = \begin{array}{ll} \rho_\perp(\tilde{B}\theta_1,\, S) \odot \cdots \odot \rho_\perp(\tilde{B}\theta_k,\, S) :: \perp & \\ \textit{if } \rho_\perp(A,\, S) = \theta_1 :: \cdots :: \theta_k :: \perp & \end{array}$$

$$\rho_\perp(G,\, S) = \begin{array}{ll} \lambda & \\ \textit{if } \rho_\perp(A,\, S) = \lambda. & \end{array}$$

We may define a function $\phi_P : \mathcal{B}^*_E \mapsto \mathcal{B}^*_E$ which, given the abstraction of a frontier, returns the abstraction of a subsequent one.

Definition 6.29 *[17] Let* P *be the program* $c_1 :: \cdots :: c_n$. $\phi_P : \mathcal{B}^*_E \mapsto \mathcal{B}^*_E$ *is defined clause-wise as the concatenation* $\phi_P(S) = \phi_{c_1}(S) :: \cdots :: \phi_{c_n}(S)$, *for any sequence* S. *Let* c *be a clause standardized apart from* S. *We distinguish two cases, for unit and non-unit clauses.*

- *If c is the unit clause $A.$, then* $\phi_{A.}(S) = A$.
- *Otherwise, let* $c = A\,\texttt{:-}\,B, \tilde{D}$ *and* $S = d_1 :: \cdots :: d_k$. *Then*
  $\phi_c(S) \;=\; \alpha_1 :: \cdots :: \alpha_k$

$$\textit{where } \alpha_i = \left| \begin{array}{lll} ?A\theta_i & \textit{if } d_i = ?B' & \textit{and } \theta_i = mgu(B, B'), \\ \phi_{(A\,\texttt{:-}\,\tilde{D})\theta_i}(S) & \textit{if } d_i = B' & \textit{and } \theta_i = mgu(B, B'), \\ \lambda & \textit{otherwise.} & \end{array} \right.$$

Note that ϕ_P is an abstract version of the unfolding operator applied to sequences.

Interpretations are elements of the complete lattice $(\mathcal{P}(\mathcal{B}_E^*), \subseteq, \bot, \top, \cup, \cap)$, i.e. sets of sequences representing abstractions of frontiers. The immediate consequences operator Φ_P extends ϕ_P to interpretations.

Definition 6.30 *[17] The immediate consequences operator* $\Phi_P : \mathcal{P}(\mathcal{B}_E^*) \mapsto \mathcal{P}(\mathcal{B}_E^*)$ *is defined in terms of* ϕ_P *as follows. Let* $I \in \mathcal{P}(\mathcal{B}_E^*)$,

$$\Phi_P(I) \quad = \quad \{\phi_P(S) \mid S \in I\} \;\cup\; \{P^\sharp\}.$$

Φ_P is continuous on the lattice of interpretations and the fixpoint semantics $\mathcal{S}_{DFL}(P)$ defined in [17] is its least fixpoint. It contains a possibly infinite set of abstractions of increasing frontiers.

$\mathcal{S}_{DFL}(P)$ has been defined by considering most general goals. According to the s-semantics style, it encodes the information on any goal. To extract this information we use the reachability function $\rho_\bot$. Any Prolog answer substitution (p.a.s.) for a goal G in the program P can be characterized in terms of the reachability of G in one of the sequences in $\mathcal{S}_{DFL}(P)$.

Theorem 6.31 *[17] Let G be a goal and P be a program. θ is a p.a.s for G in P if and only if there exists $S \in \mathcal{S}_{DFL}(P)$ such that $\theta \;\in\; \rho_\bot(G, S)$.*

Therefore $\mathcal{S}_{DFL}(P)$ is correct with respect to $\approx_{p.a.s.}$. Actually, the idea behind the definition of reachability is to capture also other issues involved in the computation of a Prolog answer substitution such as sequences of answers and termination. In fact, the analogous of theorem 5.17 holds for the Prolog search strategy, i.e. when p.a.s. instead of c.a.s. are considered.

Theorem 6.32 *[17] Let G be a goal and P be a program. Then*

> *G universally terminates with p.a.s. $\theta_1, \ldots, \theta_n$ iff there exists $S \in \mathcal{S}_{DFL}(P)$ such that $\rho_\bot(G, S) = \theta_1 :: \ldots :: \theta_n$.*
>
> *G has an infinite computation iff for every $S \in \mathcal{S}_{DFL}(P)$ $\rho(G, S) = s ::\bot$ for some sequence s of p.a.s for G.*

There are analogies between $\mathcal{S}_{DFL}(P)$ and other functional semantics for PROLOG developed in the denotational style. For instance, in [14] the semantics is a function which associates to any goal an extended or infinite sequence of p.a.s. which clearly recalls the sequence computed by $\rho_\bot$. The difference is in the style of the semantics construction. The semantics according to the functional style is a function defined as the (least) solution of a given recursive set of equations. The semantics defined according to the the s-semantics approach is instead a syntactic object, which encodes information on the observable, collected in a goal independent way.

Another semantics which can be viewed as an abstraction of $\mathcal{O}^{\mathcal{R}}_{\leftarrow}(P)$ is presented in [11]. The sequence of clauses is abstracted by its length, while the solution to the control problem of Prolog is solved by resorting to a notion of *oracle*, which defines, at each computation step, the set of clauses applicable to rewrite the current resolvent. The use of the oracle induces an elegant semantics characterization in which the logical and control components of Prolog are dealt with independently. The logical reading of a program results thus unaffected. The program's semantics is defined parametrically on the oracle. This gives to the approach a quite general flavour. The semantics in [11] has only a top-down definition. However, a more recent version of [11] contains two equivalent (top-down and bottom-up) semantics much in the style of $\mathcal{O}^{\mathcal{R}}_{R}(P)$ and more similar to the semantics in [17].

Other extensions of the s-semantics approach which are not related to the frontiers semantics defined in section 5.3 are presented in [6, 10].

In [10] a compilative approach to model Prolog control is defined. Instead of collecting information concerning the control of the program in the semantics, the program itself is enhanced so that its standard meaning reflects the required control. A logic program P is transformed into a program P^{τ} defined on a constraint language which contains ask-tell constrained clauses. Ask constraints are interpreted by an associated termination theory which captures the control of a Prolog program.

In [6] various Prolog built-in's that include arithmetic operations and metalogical relations like *var* and *ground* are considered. Only the Prolog leftmost selection rule is taken into account. Interpretations are sets of pairs $\langle A, \eta \rangle$, where A is an atom and η is a substitution whose domain is contained in the set of variables occurring in A. η is meant to represent a computed answer substitution for the goal A. Suitable notions of truth and model are defined on these interpretations and the existence of a least model is shown. The primitive predicates considered in [6] are called first-order built-in's to distinguish them from those built-in's which refer to clauses and goals like *call*. In [97] this second class of built-in's is considered.

6.6 Modular logic programs with inheritance

As already mentioned, by modifying $\mathcal{O}_{\Omega}(P)$ we can obtain semantics which are compositional w.r.t. other composition operators. In this section we will show an extension of such a semantics introduced in [18] to model several inheritance mechanisms in a compositional way.

In [18] inheritance is viewed as a mechanism for differential programming, i.e. a mechanism for constructing new program components by specifying how they differ from the existing ones. Differential programming is achieved by using "filters" to modify the external behavior of existing components. Accordingly, a modified version of a component is obtained by defining a new component that performs some special operations and possibly calls the original one. An intuitive justification for such an interpretation can be found in [35]. See also [26] for a survey on inheritance mechanisms in logic programming.

Differential programs [18] are program components, i.e. logic programs annotated by three sets of *exported* predicate symbols (the external interface):

Σ: statically inherited predicates (à la *Simula-67*);

Δ: dynamically inherited predicates (à la *Smalltalk*);

Θ: extensible predicates.

The three sets are mutually disjoint and their union is contained in the set $\pi(P)$ of the predicate symbols occurring in P. The remaining predicates, $\pi(P) \setminus (\Sigma \cup \Delta \cup \Theta)$ will be henceforth referred to as *internal* predicates and denoted by $\iota(P)$.

Similarly to classes in the O-O paradigm, differential programs can be organized in *isa* hierarchies and can use inherited definitions according to their external interfaces. Intuitively, in a hierarchy P *isa* Q the unit P can *inherit* some of the classes and some of the methods defined by the unit Q. Statically and dynamically inherited predicates are evaluated according to an overriding semantics. The distinction between the two sets Σ and Δ reflects the distinction between two different forms of inheritance (static and dynamic respectively). The idea is that a differential program P is to be understood as part of a structured context of the form C *isa* P *isa* D and that the evaluation of a goal depends on the annotation of its predicate symbols. A Σ-predicate is evaluated in P using P's local definition or any definition inherited from the context D. The local definition, if there is any, overrides the inherited one. Hence, any occurrence in P of a goal for a static predicate which is also defined in P is bound to the local definition independently of the context in which P occurs. Conversely, the evaluation of a Δ-predicate in P uses the local definition or the inherited one, only if no definition for the same predicate name is provided by the context C. If C contains a definition, then this definition overrides in P the local or inherited one.

The annotation Θ models an orthogonal composition mechanism defined according to an *extension* semantics whereby local definitions are extended by inherited ones. Therefore, the definition of a Θ-predicate in P can be *extended* (not overridden) by the definitions in C and in D.

The *isa* specialization operator should be thought of as right-associative, i.e. the hierarchy P_n *isa* P_{n-1} *isa* $\cdots$ *isa* P_1 is to be understood as P_n *isa* $(P_{n-1}$ *isa* $(\cdots$ *isa* $(P_2$ *isa* $P_1)\cdots))$. The following example shows the use of these composition mechanisms.

Example 6.33 *[18] Consider two classes* **Student** *and* **CS_Student** *(computer science student).* **CS_Student** *is a subclass of* **Student** *and redefines one of its superclass' methods. The two classes can be defined as differential logic programs as follows.*

CS_Student	*isa*	**Student**
$whoAmI(aCS\_Student)$.		$whoAmI(aStudent)$.
		$whoAreYou(X)$:-$whoAmI(X)$.
$address(theCS\_Dept)$.		$address(univ\_hall)$.
		$adm\_addr(X)$:-$address(X)$.
		$course(X)$:-$required(X)$.
$required(logicProg)$.		$required(4thLevel)$.
.....		....

where, in both **Student** *and* **CS_Student**, whoAmI *and* whoAreYou *are annotated as* Δ*-predicates,* address *and* adm_addr *as* Σ*-predicates,* course *and* required *as* Θ*-predicates. The use of different annotations for the exportable predicates of the two programs is motivated by the behavior we expect in response to the different queries for the hierarchy* **CS_Student** *isa* **Student**. *Consider first the query* whoAreYou(X). *Here, the expected answer is* $X/aCS\_Student$ *and can be obtained by taking* whoAmI *to be a* Δ*-predicate. Note that* **CS_Student** *inherits the definition of* whoAreYou *from* **Student** *and, since* whoAmI *is a* Δ*-predicate, the evaluation of the call* whoAmI(X) *uses the definition contained in* **CS_Student**. *Consider now the query* adm_addr(X). *Here, the expected answer is* $X/univ\_hall$ *because we assume that the administrative address of a student is independent of the department where that student belongs. This behavior can be modeled by defining* address *to be a* Σ*-predicate. This guarantees that the evaluation of the call* address(X) *uses the definition local to* **Student**.

Finally, we can model the fact that a **CS_Student** *is expected to take all of the courses required for a* **Student** *by defining* course *and* required *to be* Θ *predicates.*

The operational semantics for hierarchies is formally given by defining a suitable inference rule $\vdash$ obtained by modifying SLD-resolution to take into account the inheritance mechanisms expressed by the *isa* construct. $HP \vdash_\theta G$ denotes the derivation of the goal G in the hierarchy HP with computed answer θ. Two *isa*-hierarchies HP and HP' are observationally equivalent ($\approx_{isa}$) with respect to answer substitutions if for every goal G and every substitution θ, $HP \vdash_\vartheta G$ iff $HP' \vdash_{\vartheta'} G$, and $\vartheta_{|var(G)} = \vartheta'_{|var(G)}$.

The corresponding observational equivalence $\approx_{diff}$ for differential programs is defined as

Definition 6.34 *Let P_1, P_2 be differential programs. $P_1 \approx_{diff} P_2$ if for every differential program Q and for every hierarchy HP*

$$Q \ isa \ (P_1 \ isa \ HP) \approx_{isa} Q \ isa \ (P_2 \ isa \ HP).$$

In order to obtain a compositional semantics for *isa* hierarchies, a syntactic composition operator $\triangleleft$ on programs has been introduced in [18]. Such an operator makes it possible to translate an *isa* hierarchy $HP = P_n \ isa \ldots isa \ P_1$ into an equivalent "flat" program $HP_\triangleleft = P_n \triangleleft \ldots \triangleleft P_1$ to be evaluated by standard SLD-derivation. The next theorem shows the equivalence between the $\vdash$ derivations in HP and SLD-derivations (denoted by $\rightsquigarrow$) in $HP_\triangleleft$.

Theorem 6.35 *[18] Let $HP = P_n \ isa \ldots isa \ P_1$ be an isa-hierarchy and $HP_\triangleleft = P_n \triangleleft \ldots \triangleleft P_1$ be the corresponding $\langle \Sigma, \Delta, \Theta \rangle$-differential program. Then for any goal G such that $Pred(G) \subseteq (\Sigma \cup \Delta \cup \Theta)$*

$$HP \vdash_\theta G \quad \Longleftrightarrow \quad G \overset{\gamma}{\rightsquigarrow}_{HP_\triangleleft} \square$$

and $\gamma_{|var(G)} = \vartheta_{|var(G)}$.

For the sake of simplicity we do not give here the formal definition of $\triangleleft$ (which essentially uses renamings to simulate the overriding mechanisms of dynamic and static predicates). However it is worth noting that, according to the correspondence with *isa* hierarchies stated by the previous theorem, such an operator allows us to capture several specialized mechanisms such as static and dynamic inheritance and composition by union of clauses. The following is just an example of program composition obtained by using $\triangleleft$.

Example 6.36 *Consider the two programs* **CS_Student** *and* **Student** *as defined in example 6.33 with $\Sigma = \{address, adm\_addr\}$, $\Delta = \{whoAmI,$ $whoAreYou\}$ and $\Theta = \{course, required\}$. Then the $\triangleleft$ composition of the programs is given by:*

CS_Student $\triangleleft$ Student

$whoAmI(aCS\_Student).$	
	$whoAreYou(X) \text{:-} whoAmI(X).$
$address(theCS\_Dept).$	$s\_address(univ\_hall).$
	$s\_adm\_addr(X) \text{:-} s\_address(X).$
	$course(X) \text{:-} required(X).$
$required(logicProg).$	$required(4thLevel).$

Note that the evaluation of the goal $whoAreYou(X)$ in the program **CS_Student** $\lhd$ **Student** *by using SLD-derivation produces the answer $X/aCS\_Student$ while the query $adm\_addr(X)$ gives the answer $X/univ\_hall$, which corresponds to the answers obtained by using $\vdash$ in* **CS_Student** *isa* **Student**.

A fixpoint semantics, compositional with respect to the $\lhd$ operator and correct with respect to $\approx_{diff}$, has been obtained by a generalization of the semantics $\mathcal{O}_\Omega(P)$ of section 4. The next example shows that $\mathcal{O}_\Omega(P)$ does not contain enough information to model the program composition we are considering. Hence the generalization is truly necessary.

Example 6.37 *[18] Let $\langle\Sigma_1,\Delta_1,\Theta_1\rangle$-$P_1$ and $\langle\Sigma_2,\Delta_2,\Theta_2\rangle$-$P_2$ be the programs*

$$P_1 = \{r(a).\} \qquad P_2 = \{\ \begin{array}{l} p(X) : -r(X). \\ r(b). \end{array} \ \},$$

where $\Delta_1 = \{r\}$, $\Delta_2 = \{r,p\}$ and $\Sigma_i = \Theta_i = \emptyset$ for $i = 1,2$. The composition $P_1 \lhd P_2$ corresponds to the program $\{r(a).,\ p(X)\text{:-}r(X).\}$ where the clause $r(b) \in P_2$ has been overriden by the clause $r(a) \in P_1$. According to the definition of the $\mathcal{O}_\Omega$ semantics we have

$$\mathcal{O}_\Omega(P_1 \cup P_2) = \{r(b), p(b), r(a), p(a), p(X) : -r(X)\}$$

In order to obtain the semantics of $P_1 \lhd P_2$, we should then delete from $\mathcal{O}_\Omega(P_1 \cup P_2)$ not only $r(b)$, which is an obvious consequence of the overriding semantics of $\lhd$, but also everything derived from $r(b)$ ($p(b)$ in this case). Thus, when defining the semantics of P_2, we need a mechanism for recording that $p(b)$ has been obtained by using the definition of the Δ-predicate r, local to P_2, which could be overridden by the context.

The problem shown by the previous example is solved by introducing *context sensitive clauses* as elements of the semantic domain.

Definition 6.38 *[18] A* context sensitive clause (cs-clause) *is an object of the form*

$$A : -\{q_1, \ldots, q_n\} \Box B_1, \ldots, B_k \tag{1}$$

where $q_1, \ldots, q_n$ are predicate symbols.

The intuitive meaning of (1) is that the logical implication $A \leftarrow B_1, \ldots, B_k$ is true in any context which does not override the definitions of $q_1, \ldots, q_n$. A standard clause can be viewed as a cs-clause with an empty set of names. The equivalence $\approx_+$ on clauses (definition 4.5) naturally extends to cs-clauses. Let $\mathcal{C}_\Delta$ be the set of all the equivalence classes of cs-clauses $A : -s \Box \tilde{B}$ such that $s \subseteq \Delta$. A cs-interpretation I for a $\langle\Sigma,\Delta,\Theta\rangle$-program P is any $I \subseteq \mathcal{C}_\Delta$.

The fixpoint semantics of differential programs is given in terms of an immediate consequences operator for *cs*-interpretations, T_P^{cs}, and this, in turn, can be simply defined in terms of a modified unfolding rule $unf_{P,\Psi,\Delta}$. Let P be a $\langle\Sigma,\Delta,\Theta\rangle$-program and $\kappa(P)$ be the set of predicates defined in P. The set of predicates whose definitions can be modified by composing P is the set (open predicates) $Open(P) = (\Sigma \setminus \kappa(P)) \cup \Delta \cup \Theta$.

Definition 6.39 *[18] Let P be a $\langle\Sigma,\Delta,\Theta\rangle$-program and let I be a cs-interpretation for P. Then*

$$T_P^{cs}(I) = unf_{P,Open(P),\Delta}(I \cup Id_{Open(P)}).$$

where, given two sets of predicate names Ψ and Δ,

$$
\begin{array}{lll}
unf_{P,\Psi,\Delta}(I) = \{ & A\theta : \text{-} s \cup C \cup C_1 \ldots \cup C_k \ \square \ (\tilde{L}_1, \ldots, \tilde{L}_k)\theta & \mid \\
& \exists\ A : \text{-} s \ \square \ B_1, \ldots, B_k \in P, & \\
& \exists\ cl_i = B'_i : \text{-} C_i \square \tilde{L}_i, i = 1, \ldots, k, \ \textit{variants} & \\
& \quad \textit{of cs-clauses in } I \cup Id_\Psi \ \textit{ and renamed apart}, & \\
& \theta = mgu((B_1, \ldots, B_k), (B'_1, \ldots, B'_k)), & \\
& C = \{Pred(B_i) \ \mid \ Pred(B_i) \in \Delta \textit{ and } cl_i \notin Id_\Psi\} & \}
\end{array}
$$

It is worth noting that when all the clauses in the cs-interpretations have empty sets of predicate names, the previous operator is exactly the operator defined in definition 4.9. Moreover, when cs-interpretations contain unit clauses only and $Open(P) = \emptyset$, the previous definition boils down to the operator of definition 3.16.

T_P^{cs} is continuous on $(\mathcal{C}_\Delta, \subseteq)$. Hence the fixpoint semantics is the following.

Definition 6.40 *[18] Let P be a $\langle \Sigma, \Delta, \Theta \rangle$-program. The fixpoint semantics $[\![P]\!]$ of P is defined as*

$$[\![P]\!] = T_P^{cs} \uparrow \omega \setminus (A \cup B)$$

where

$$
\begin{array}{lll}
A = \{H : -s \square \tilde{B} \mid & Pred(H) \in \iota(P)\} & \textit{and} \\
B = \{H : -s \square \tilde{B} \mid & \exists H' : -s' \square \tilde{B}' \in T_P^{cs} \uparrow \omega \textit{ such that} & \\
& s' \subset s, H' : -\tilde{B}' \approx_+ H : -\tilde{B} & \}
\end{array}
$$

We refer to [18] for the details on the previous construction and in the following we will only show the main results which hold for the $[\![P]\!]$ semantics. Compositionality of $[\![P]\!]$ wrt $\lhd$ has been proven by introducing a (right associative) semantic operator $\prec$ on cs-interpretations which corresponds to the syntactic composition $\lhd$ of differential programs.

Theorem 6.41 (compositionality) *[18] Let $\langle \Sigma_P, \Delta_P, \Theta_P \rangle$-$P$ and $\langle \Sigma_Q, \Delta_Q, \Theta_Q \rangle$-$Q$ be differential programs. Then*

$$[\![P \lhd Q]\!] \ = \ [\![P]\!] \prec [\![Q]\!].$$

Note that, according to the previous remarks, the $[\![P]\!]$ semantics has as an instance (for the case $Open(P) = \emptyset$) the s-semantics. Therefore, by using the correctness of the s-semantics, it is easy to show that $[\![P]\!]$ correctly models computed answers. By exploiting the correspondence between $\lhd$ composition and *isa* hierarchies (theorem 6.35) and the compositionality (theorem 6.41) we can then obtain the following result which shows that the computed answers of an *isa* hierarchy can be obtained, in a compositional way, from the semantics of the components of the hierarchy.

Theorem 6.42 *[18] Let $HP = P_n \ isa \ldots isa \ P_1$ be an isa-hierarchy, $HP_\lhd = P_n \lhd \ldots \lhd P_1$ be the corresponding $\langle \Sigma, \Delta, \Theta \rangle$-program and $G = A_1, \ldots, A_k$ be a goal with $Pred(G) \subseteq (\Sigma \cup \Delta \cup \Theta)$. Then*

$$
\begin{array}{lll}
HP \vdash_\vartheta G \quad \Longleftrightarrow \quad & \exists\ H_i : -s_i \ \square \in [\![P_n]\!] \prec \ldots \prec [\![P_1]\!], & \\
& i = 1, \ldots, k, & \\
& \exists \gamma = mgu((A_1, \ldots, A_k)(H_1, \ldots, H_k)), & \\
& \gamma_{|var(G)} = \vartheta_{|var(G)}. &
\end{array}
$$

In terms of observational equivalences, we have the following result which shows the correctness of $[\![P]\!]$ wrt $\approx_{diff}$.

Corollary 6.43 (correctness)*[18] Let P_1 and P_2 be two differential programs. Then*

$$[\![P_1]\!] = [\![P_2]\!] \;\Rightarrow\; P_1 \approx_{diff} P_2.$$

It is worth noticing that this semantics is the first compositional semantics of units and inheritance which correctly models computed answer substitutions.

7 Applications

As already mentioned, the main motivation of the s-semantics approach is to provide a semantics useful for program analysis and transformation. There exist already several applications which show that this is really the case.

7.1 Program transformation

A main concern when transforming a program is the preservation of its *semantics*. When this is the case, the transformation is called *safe*. However a transformation can be safe with respect to one semantics but not with respect to another one. For instance, in the program

$$\{\; p(X) : -q(X), q(X). \qquad q([a, Y]). \qquad q([Z, b]).\;\}$$

the duplicated atom $q(X)$ in the first clause is superfluous when considering the least Herbrand model semantics and then it can be safely deleted from the body of the clause. The same operation is not safe when the computed answers semantics is considered. In fact the answer substitution $X/[a, b]$ would be missed in the transformed program.

As a matter of fact, all the program transformation techniques, such as unfold/fold [107] and partial evaluation [78], are defined so as to preserve some observational equivalences. In most of these techniques, the relevant observables are computed answers (and sometimes finite failures). There exists at least one technique, the partial evaluation of "open" programs [111, 106, 86], whose aim is to preserve a $\cup$-compositional program equivalence[8].

Most of the transformation techniques are proved to be safe w.r.t. the declarative semantics only, thus failing to capture the safeness w.r.t. the more complex observable behavior. In some cases the observational equivalences related to computed answers [76, 89] and to finite failures [102] are considered. Usually proving that the transformation preserves the observational equivalence is rather complex (see, for example, the proofs of the partial evaluation theorems in [89]). The same goal could more easily be achieved by proving that the transformation preserves a semantics which correctly models the relevant observable. The proof can in fact be based on general theorems (such as AND-compositionality) and on powerful technical tools such as the specialized immediate consequences operators. This is the approach taken in [19] and [7], where the reference semantics are the answer substitution semantics and the semantic kernel respectively.

In [19] some transformation operations which are basic for all the transformation techniques for logic programs, such as partial evaluation, program specialization, program synthesis and optimization, are considered. For each operation, *applicability* conditions which guarantee the safeness of the trasformation with respect to the s-semantics of section 3 are defined. Not surprisingly, unfolding does not need any applicability condition. All the other operations, if not correctly applied, may lead to undesiderable observable behaviors. With only one exception, the

[8] The $\cup$-compositional semantics of section 4 is essentially the result of the *partial evaluation*, where derivations terminate at open predicates (i.e. predicates in Ω).

s-semantics of a given program contains enough information to characterize correct transformations. In fact all the applicability conditions are given in terms of properties of the s-semantics of the program to be transformed. The only exception is the folding operation. Safeness of folding cannot be ensured by just inspecting the s-semantics as the following example shows.

Example 7.1 *Consider the following program.*

$$\begin{array}{ll} P = \{ & p : -r., \quad r : -q., \quad q. \quad \} \\ \mathcal{O}(P) = & \{p, q, r\} \qquad\qquad\qquad\quad , \end{array}$$

The definition $p \equiv q$ is consistent [19] with P, since both p and q belong to $\mathcal{O}(P)$, but, if we use it to fold the body of the second clause we obtain

$$P' = \{ \quad p : -r., \quad r : -p., \quad q. \quad \}$$

which is by no means equivalent to the previous program. In fact $\mathcal{O}(P') = \{q\}$.

This problem has been partially overcome in [20] where a notion of *semantic delay* between atoms is introduced to give applicability conditions for folding. Semantic delay is not properly a property of the s-semantics, rather it depends on its fixpoint construction.

Turning to normal logic programs, [7] gives a very elegant proof of the correctness of unfold/fold w.r.t. several non-monotonic semantics (as, for example, the stable model and the well-founded model semantics), by showing that it preserves the semantic kernel considered in section 6.3.

7.2 Program analysis

In the area of *program analysis*, the s-semantics has been used as a foundation of several frameworks for *abstract interpretation* [13, 65, 77, 31]. Abstract interpretation is inherently semantics sensitive and different semantic definition styles lead to different approaches to program analysis. In the case of logic programs (see [38] for a broad overview), two main approaches exist, namely the *top-down* and the *bottom-up* ones [94]. The most popular approach is the top-down one, which propagates the information as SLD-resolution does. In this class there are ad-hoc algorithms, frameworks based on an operational semantics, and frameworks based on a denotational semantics. The bottom-up approach propagates the information as in the computation of the least fixpoint of the immediate consequences operator T_P. The idea of bottom-up analysis was first introduced in [93]. The main difference between the top-down and the bottom-up approach is usually related to *goal dependency*. In particular, a top-down analysis starts with a specific goal, while the bottom-up approach determines an approximation of the success set which is goal independent. As we will argue later, the application of the s-semantics approach to abstract interpretation shows that the real issue is goal dependency vs. goal independency rather than top-down vs. bottom-up. Another relevant feature of the analysis method is its ability to determine *call pattern* information [24, 73, 94], i.e. information about the procedure calls (atoms selected in an SLD-derivation). The ability to determine call patterns is also usually associated to goal dependent top-down methods. Again, the s-semantics approach shows that the choice of an adequate (concrete) semantics allows us to determine goal independent information on the call patterns and that this information can be computed both top-down and bottom-up.

The s-semantics approach to abstract interpretation was started by defining a framework for bottom-up abstract interpretation [13] based on the concrete semantics of section 3, which

correctly models computed answer substitutions. An instance of the framework consists in the specialization of a set of basic abstract operators, i.e. *abstract unification*, *abstract substitution application* and *abstract union*. Instances have been defined for ground dependency analysis [13], type inference [12] and for analysis of properties related to AND-parallelism [67, 68]. The emphasis in [13] is on the bottom-up definition of an abstract model, i.e. a goal independent approximation of the concrete denotation. Early attempts [93, 94] of defining bottom-up abstract interpretations based on the immediate consequences operators corresponding to the least Herbrand model semantics or to the semantics in [30] failed on non-trivial analyses (like mode analysis). In fact, the corresponding concrete semantics do not contain enough information on the program behavior, i.e. they are too abstract to be useful to capture program properties like variable sharing or ground dependencies.

The overall abstract interpretation methodology can be described as follows.

- Select an observable o, such that the property to be considered by the analysis is an abstraction $\alpha(o)$ of o.
- According to the s-semantics approach, select a concrete semantics $\mathcal{O}_o$ correct w.r.t. o. $\mathcal{O}_o$ can equivalently be determined by
  - A top-down construction, obtained by collecting the observables for the atomic goals of the form $p(\tilde{X})$.
  - A bottom-up construction, obtained by computing the least fixpoint of an immediate consequences operator.
- Define a suitable abstraction $\mathcal{O}_{\alpha(o)}$ of $\mathcal{O}_o$, by providing the abstract versions of the operators involved in the top-down and bottom-up definitions and by proving the correctness theorems. If the abstraction satisfies suitable properties [70, 66], we have two equivalent methods for computing the goal independent abstract denotation $\mathcal{O}_{\alpha(o)}(P)$ of the program P.
- The result of the analysis for a specific goal G can be determined by exploiting the AND-compositionality property of all the semantics defined by the s-semantics approach, including their abstract versions. Namely, the result can be obtained by *executing* G *in* $\mathcal{O}_{\alpha(o)}(P)$.

Let us discuss some specific analysis problems in the framework of the above methodology.

- If we are interested in properties of the answer substitutions (such as aliasing and sharing) we have to choose a concrete semantics correct w.r.t. answer substitutions. Therefore the least Herbrand model semantics is not adequate and a semantics at least as detailed as the one in section 3 has to be chosen.
- If we want to perform analysis of program components in a modular way, we need a semantics compositional w.r.t. program union. As a matter of fact the framework in [13] has been extended to handle modularity [32], by replacing the s-semantics with its compositional version (the Ω-semantics of section 4), which has clauses as semantic objects. This extension requires a notion of *abstract program* and a uniform treatment of concrete and abstract objects (i.e. programs and π-interpretations). The abstract meaning of a module is the result of the module analysis. The result of the analysis for the composition of the modules is obtained by composing the module abstract meanings. The extension introduces several technical complications in the abstract semantics construction dealing with

termination and space complexity. Namely an additional layer of abstraction (obtained by applying fixpoint acceleration techniques) is needed to provide finitary descriptions for arbitrary large clauses (and therefore to ensure termination), thus introducing a further approximation which makes the analysis less precise.

While this is needed to handle generic (possibly infinite) abstract domains, there exists [54] a wide class of *compositionally tractable* abstract domains (e.g *Sharing* [70] and *Prop* [36]) for which a finite description of the compositional abstract semantics can be obtained without a further level of abstraction. In fact, when considering compositionally tractable domains we are essentially considering the Ω-semantics over a finite function free signature. As shown in [54], by imposing such a restriction we can always obtain a finite characterization of the compositional semantics. This result can be applied also to the abstraction of other semantics consisting of sets of clauses, as for example the resultants semantics in [61, 59].

- If we want to determine abstract properties of the call patterns, we should use a concrete semantics which gives more information on the computation than just the computed answers. Namely, we have to model an observable consisting of all the procedure calls. The problem of analyzing properties of the call patterns has been considered in [53], where the concrete semantics is the *call patterns semantics* derived according to a local selection rule, as defined in [61, 59]. The resulting abstract semantics are goal independent, parametric w.r.t. the (local) selection rule and allow us to characterize properties of the *correct call patterns* [61, 59], which are those call patterns which belong to successful derivations.

  A similar (yet goal dependent) result can be obtained by using a transformational approach [31, 99]. A program P and a goal G are transformed (by using a transformation similar to the *magic set* transformation) into a program P', such that every call pattern of G in P is a success pattern of P'. An abstraction of the operator $T^{\Pi}_{P'}$ of definition 3.13 can now be used to compute in a bottom-up way information on the call patterns of G in P. Recently, the approach was made goal independent [33], by using the Ω-semantics. The result is a denotation consisting of clauses very similar to the one in [53].

- It is worth noting that the top-down operational or denotational frameworks [24, 94, 104, 74] do indeed contain a lot of information on the "internal" computation details. By choosing a semantics like the one of section 5.3, we can model the same observables and still get a goal independent top-down and bottom-up construction of the abstract model.

When applied to CLP, the above approach leads to a framework where abstraction simply means abstraction of the constraint system. The construction is based on a generalized algebraic semantics[9] [66], defined in terms of a constraint system and a general (constraint system independent) notion of denotation, which is as usual characterized both top-down and bottom-up. Different abstract semantics can be defined by choosing suitable abstract constraint systems. The main new result is that abstract interpretation, i.e. the construction of an abstract denotation, can be viewed as computation in a suitable instance of the same CLP framework, where the program is transformed into an *abstract program*, obtained by abstracting the concrete constraints. A similar result, in a framework based on the generalization of the top-down

[9] This semantics generalizes the approach in [39] which gives an algebraic description of a class of fixpoint semantics (including ground and non-ground concrete semantics, and various abstract semantics) in terms of abstract notions of "instance" and "normalization".

operational semantics, is described in [34]. The ability to use the CLP interpreter to analyze CLP programs has been exploited in some interesting applications [8].

7.3 Declarative debugging

The application of the s-semantics approach to semantics-based (declarative) debugging [16] has the following features when compared to the existing methods [103, 49, 87].

- The s-semantics, when taken as specification of the intended semantics, allows us to obtain a more accurate diagnosis than the one that can be obtained using the least Herbrand model or the c-semantics (which is used in [49]).
- The properties of the s-semantics (equivalent top-down goal independent denotations and bottom-up denotations) make possible to devise new elegant and powerful diagnosis methods. In particular, the top-down diagnosis can be based on the execution of atomic goals of the form $p(\tilde{X})$.
- The relation between concrete and abstract semantics, allows us to consider abstract declarative debugging, where the intended semantics is an abstraction of the concrete semantics. The intended semantics is usually represented by an *oracle* [103], which tells us whether a given object belongs to the semantics. Since abstract denotations are finite, they can explicitely be used as oracles. Then we can test a program in a uniform way w.r.t. different specifications of the program properties.

7.4 Metaprogramming

We consider here a formalization of metaprogramming [85] with the non-ground metalevel representation of object level variables. In the case of the *vanilla* metainterpreter, let P be a program and P_M be its non-ground metalevel representation. The problem is that there exists no one-to-one correspondence between the semantics of P and the semantics of $V_P = vanilla \cup P_M$. The problem is related to differences in the languages used at the metalevel and at the object level and was solved either by considering typed programs [69], or by considering language independent programs only [37]. If we consider the s-semantics of P and V_P, due to the property stated by theorem 3.11, the language problem disappears and we can easily prove the following theorem.

Theorem 7.2 *[85, 96] Let P be a positive program and V_P be its* vanilla *metainterpreted version, where the proof procedure is defined by the relation* demo. *Then, for every n-adic predicate symbol p in P,*
$demo(p(t_1, \ldots, t_n)) \in \mathcal{O}(V_P)$ *iff* $p(t_1, \ldots, t_n) \in \mathcal{O}(P)$.

A similar result was also proved [85] for a metainterpreter defining the inheritance mechanism described in [18].

8 Conclusions

We have shown several semantics, which exhibit similar properties and which are all defined according to the same methodology. We have also shown that at least some of the above semantics have successfully been used to solve real problems.

As shown in [59, 52], the various semantics are mutually related by means of abstractions. The same relation holds between concrete and abstract semantics. In particular, the generalized semantics of CLP in [66, 65] shows that one can derive from a single semantics several specializations obtained by *abstracting the constraints in the program*.

One interesting open research problem, which is currently under investigation, is whether the approach can be extended to cope with the various concrete observables. One could start with a program which has as regular semantics the most concrete one (for example, a semantics similar to the one considered in section 5.3). Such a semantics should have the usual top-down and bottom-up definitions. Moreover, the usual s-semantics theorems (AND-compositionality, correctness w.r.t. the observable, equivalence of the two definitions) should hold. All the other (concrete and abstract) semantics should then be derivable simply by abstracting the constraints in the program, thus obtaining for free the validity of all the theorems, once the correctness of the abstraction on the constraint system has been proved. The theory should also allow us to discuss in general terms of properties such as the independence from the selection rule, the $\cup$-compositionality and the full abstraction.

References

[1] M. Alpuente and M. Falaschi. Narrowing as an Incremental Constraint Satisfaction Algorithm. In J. Maluszyński and M. Wirsing, editors, *Proc. of PLILP'91*, volume 528 of *Lecture Notes in Computer Science*, pages 111–122. Springer-Verlag, Berlin, 1991. Extended version to appear in *Theoretical Computer Science*.

[2] M. Alpuente, M. Falaschi, M. Gabbrielli, and G. Levi. The semantics of equational logic programming as an instance of CLP. In K. R. Apt, J. W. de Bakker, and J. J. M. M. Rutten, editors, *Logic Programming Languages: Constraints, Functions and Objects*, pages 49–81. The MIT Press, Cambridge, Mass., 1993.

[3] K. R. Apt. Introduction to Logic Programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 495–574. Elsevier, Amsterdam and The MIT Press, Cambridge, 1990.

[4] K. R. Apt, H. Blair, and A. Walker. Towards a Theory of Declarative Knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann, Los Altos, Ca., 1988.

[5] K. R. Apt and M. Gabbrielli. Declarative Interpretations Reconsidered. CWI, Amsterdam. Submitted for publication, 1993.

[6] K. R. Apt, E. Marchiori, and C. Palamidessi. A theory of first-order built-in's of PROLOG. In H. Kirchner and G. Levi, editors, *Algebraic and Logic Programming, Proceedings of the Third International Conference*, volume 632 of *Lecture Notes in Computer Science*, pages 69–83. Springer-Verlag, Berlin, 1992.

[7] C. Aravindan and Phan Minh Dung. On the correctness of unfold/fold transformation of normal and extended logic programs. Technical report, Asian Institute of Technology, Bangkok, Thailand, 1993.

[8] R. Bagnara, R. Giacobazzi, and G. Levi. An Application of Constraint Propagation to Data-flow Analysis. In *Proc of Ninth IEEE Conference on AI Applications*, pages 270–276. IEEE Computer Society Press, 1993.

[9] I. Balbin and K. Ramamohanarao. A Generalization of the Differential Approach to Recursive Query Evaluation. *Journal of Logic Programming*, 4:259–262, 1987.

[10] R. Barbuti, M. Codish, R. Giacobazzi, and G. Levi. Modelling Prolog Control. *Journal of Logic and Computation*, 3, 1993.

[11] R. Barbuti, M. Codish, R. Giacobazzi, and M. Maher. Oracle Semantics for PROLOG. In H. Kirchner and G. Levi, editors, *Algebraic and Logic Programming, Proceedings of the Third International Conference*, volume 632 of *Lecture Notes in Computer Science*, pages 100–114. Springer-Verlag, Berlin, 1992.

[12] R. Barbuti and R. Giacobazzi. A Bottom-up Polymorphic Type Inference in Logic Programming. *Science of Computer Programming*, 19(3):281–313, 1992.

[13] R. Barbuti, R. Giacobazzi, and G. Levi. A General Framework for Semantics-based Bottom-up Abstract Interpretation of Logic Programs. *ACM Transactions on Programming Languages and Systems*, 15(1):133–181, 1993.

[14] M. Baudinet. Proving Termination Properties of Prolog Programs: A Semantic Approach. *Journal of Logic Programming*, 14:1–29, 1992.

[15] E. Bertino, M. Martelli, and D. Montesi. CLP(AD) as a Deductive Database Language with Updates. In E. Lamma and P. Mello, editors, *Extensions of Logic Programming. Proc. Third International Workshop on Extensions of Logic Programming, ELP'92*, volume 660 of *Lecture Notes in Artificial Intelligence*, pages 80–99. Springer-Verlag, Berlin, 1992.

[16] E. Bolzan. Proprietà osservabili e diagnosi di programmi logici. Master's thesis, Dipartimento di Informatica, Università di Pisa, 1993. in italian.

[17] A. Bossi, M. Bugliesi, and M. Fabris. Fixpoint semantics for PROLOG. In D. S. Warren, editor, *Proc. Tenth Int'l Conf. on Logic Programming*, pages 374–389. The MIT Press, Cambridge, Mass., 1993.

[18] A. Bossi, M. Bugliesi, M. Gabbrielli, G. Levi, and M. C. Meo. Differential logic programming. In *Proc. Twentieth Annual ACM Symp. on Principles of Programming Languages*, pages 359–370. ACM Press, 1993.

[19] A. Bossi and N. Cocco. Basic transformation operations for logic programs which preserve computed answer substitutions of logic programs. *Journal of Logic Programming*, 16:47–87, 1993.

[20] A. Bossi, N. Cocco, and S. Etalle. On Safe Folding. In M. Bruynooghe and M. Wirsing, editors, *Programming Language Implementation and Logic Programming - Proceedings PLILP'92*, volume 631 of *Lecture Notes in Computer Science*, pages 172–186. Springer-Verlag, Berlin, 1992.

[21] A. Bossi, M. Gabbrielli, G. Levi, and M. C. Meo. Contributions to the Semantics of Open Logic Programs. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1992*, pages 570–580, 1992. Extended version to appear in *Theoretical Computer Science*.

[22] A. Bossi and M. Menegus. Una Semantica Composizionale per Programmi Logici Aperti. In P. Asirelli, editor, *Proc. Sixth Italian Conference on Logic Programming*, pages 95–109, 1991.

[23] P. Bruscoli, F. Levi, G. Levi, and M. C. Meo. Compilative Constructive negation in Constraint Logic Programs. In S. Tison, editor, *Proc. CAAP'94*, *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1994.

[24] M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming*, 10:91–124, 1991.

[25] F. Bry. Logic Programming as Constructivism: A Formalization and its Application to Databases. In *Proc. Eighth ACM Symp. on Principles of Database Systems*, 1989.

[26] M. Bugliesi, E. Lamma, and P. Mello. Modularity in Logic Programming. *Journal of Logic Programming*, 1994. To appear.

[27] D. Chan. Constructive Negation Based on the Completed Database. In R. A. Kowalski and K. A. Bowen, editors, *Proc. Fifth Int'l Conf. on Logic Programming*, pages 111–125. The MIT Press, Cambridge, Mass., 1988.

[28] D. Chan. An Extension of Constructive Negation and its Application in Coroutining. In E. Lusk and R. Overbeek, editors, *Proc. North American Conf. on Logic Programming'89*, pages 477–493. The MIT Press, Cambridge, Mass., 1989.

[29] K. H. Chan. Equivalent logic programs. *Journal of Logic Programming*, 9(3):187–199, 1990.

[30] K. L. Clark. Predicate logic as a computational formalism. Res. Report DOC 79/59, Imperial College, Dept. of Computing, London, 1979.

[31] M. Codish, D. Dams, and E. Yardeni. Bottom-up Abstract Interpretation of Logic Programs. Technical report, Dept. of Computer Science, The Weizmann Institute, Rehovot, 1990. To appear in *Theoretical Computer Science*.

[32] M. Codish, S. K. Debray, and R. Giacobazzi. Compositional Analysis of Modular Logic Programs. In *Proc. Twentieth Annual ACM Symp. on Principles of Programming Languages*, pages 451–464. ACM Press, 1993.

[33] M. Codish and B. Demoen. Analysing Logic Programs using "`prop`"-ositional Logic Programs and a Magic Wand. In D. Miller, editor, *Proc. 1993 Int'l Symposium on Logic Programming*. The MIT Press, Cambridge, Mass., 1993.

[34] P. Codognet and G. Filè. Computations, Abstractions and Constraints. In *Proc. Fourth IEEE Int'l Conference on Computer Languages*. IEEE Press, 1992.

[35] W. Cook and J. Palsberg. A Denotational Semantics of Inheritance and its Correctness. In *Proceedings of OOPSLA'89*, pages 433–443. ACM, 1989.

[36] A. Cortesi, G. Filè, and W. Winsborough. *Prop* revisited: Propositional Formula as Abstract Domain for Groundness Analysis. In *Proc. Sixth IEEE Symp. on Logic In Computer Science*, pages 322–327. IEEE Computer Society Press, 1991.

[37] D. De Schreye and B. Martens. A Sensible Least Herbrand Semantics for Untyped Vanilla Meta-Programming and its Extension to a Limited Form of Amalgamation. In A. Pettorossi, editor, *Meta-Programming in Logic. Third International Workshop, META'92*, volume 649 of *Lecture Notes in Computer Science*, pages 192–204. Springer-Verlag, Berlin, 1993.

[38] S. K. Debray. Formal bases for dataflow analysis of logic programs. In G. Levi, editor, *Advances in logic programming theory*. Oxford University Press, 1994. To appear.

[39] S. K. Debray and R. Ramakrishnan. Generalized Horn Clause Programs. Technical report, Dept. of Computer Science, The University of Arizona, 1991.

[40] G. Delzanno and M. Martelli. A bottom-up characterization of finite success. Technical report, Università di Genova, DISI, 1992.

[41] F. Denis and J.-P. Delahaye. Unfolding, Procedural and Fixpoint Semantics of Logic Programs. In C. Choffrut and M. Jantzen, editors, *STACS 91*, volume 480 of *Lecture Notes in Computer Science*, pages 511–522. Springer-Verlag, Berlin, 1991.

[42] P. Deransart and G. Ferrand. Programmation en logique avec negation: presentation formelle. Technical Report No. 87/3, Lab. d'Informatique, Département de Mathématiques et d'Informatique, Université d'Orléans, 1987.

[43] Phan Minh Dung and K. Kanchanasut. A Fixpoint Approach to Declarative Semantics of Logic Programs. In E. Lusk and R. Overbeek, editors, *Proc. North American Conf. on Logic Programming'89*, pages 604–625. The MIT Press, Cambridge, Mass., 1989.

[44] K. Eshghi and R. A. Kowalski. Abduction compared with Negation by Failure. In G. Levi and M. Martelli, editors, *Proc. Sixth Int'l Conf. on Logic Programming*, pages 234–254. The MIT Press, Cambridge, Mass., 1989.

[45] M. Falaschi, M. Gabbrielli, G. Levi, and M. Murakami. Nested Guarded Horn Clauses. *International Journal of Foundations of Computer Science*, 1(3):249–263, 1990.

[46] M. Falaschi and G. Levi. Finite failures and partial computations in concurrent logic languages. *Theoretical Computer Science*, 75:45–66, 1990.

[47] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. Declarative Modeling of the Operational Behavior of Logic Languages. *Theoretical Computer Science*, 69(3):289–318, 1989.

[48] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. A Model-Theoretic Reconstruction of the Operational Semantics of Logic Programs. *Information and Computation*, 102(1):86–113, 1993.

[49] G. Ferrand. Error Diagnosis in Logic Programming, an Adaptation of E.Y. Shapiro's Method. *Journal of Logic Programming*, 4:177–198, 1987.

[50] M. Fitting. A Kripke-Kleene semantics for logic programs. *Journal of Logic Programming*, 2:295–312, 1985.

[51] M. Fitting and M. Ben-Jacob. Stratified and Three-valued Logic Programming Semantics. In R. A. Kowalski and K. A. Bowen, editors, *Proc. Fifth Int'l Conf. on Logic Programming*, pages 1054–1069. The MIT Press, Cambridge, Mass., 1988.

[52] M. Gabbrielli. *The Semantics of Logic Programming as a Programming Language*. PhD thesis, Dipartimento di Informatica, Università di Pisa, 1992.

[53] M. Gabbrielli and R. Giacobazzi. Goal independency and call patterns in the analysis of logic programs. In Proc. *ACM Symposium on Applied Computing*, ACM press, 1994.

[54] M. Gabbrielli, R. Giacobazzi, and D. Montesi. Modular logic programs over finite domains. In D. Saccà, editor, *Proc. Eight Italian Conference on Logic Programming*, pages 663–678, 1993.

[55] M. Gabbrielli and G. Levi. Modeling Answer Constraints in Constraint Logic Programs. In K. Furukawa, editor, *Proc. Eighth Int'l Conf. on Logic Programming*, pages 238– 252. The MIT Press, Cambridge, Mass., 1991.

[56] M. Gabbrielli and G. Levi. On the Semantics of Logic Programs. In J. Leach Albert, B. Monien, and M. Rodriguez-Artalejo, editors, *Automata, Languages and Programming, 18th International Colloquium*, volume 510 of *Lecture Notes in Computer Science*, pages 1–19. Springer-Verlag, Berlin, 1991.

[57] M. Gabbrielli and G. Levi. Unfolding and fixpoint semantics of concurrent constraint programs. *Theoretical Computer Science*, 105:85–128, 1992.

[58] M. Gabbrielli, G. Levi, and M. Martelli. New Semantics Tools for Logic Programs. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Semantics: Foundations and Applications, Proceedings REX Workshop*, volume 666 of *Lecture Notes in Computer Science*, pages 204–235. Springer-Verlag, Berlin, 1993.

[59] M. Gabbrielli, G. Levi, and M. C. Meo. Observational Equivalences for Logic Programs. In K. Apt, editor, *Proc. Joint Int'l Conf. and Symposium on Logic Programming*, pages 131–145. The MIT Press, Cambridge, Mass., 1992.

[60] M. Gabbrielli, G. Levi, and D. Turi. A Two Steps Semantics for Logic Programs with Negation. In A. Voronkov, editor, *Proceedings of the Int'l Conf. on Logic Programming and Automated Reasoning*, volume 624 of *Lecture Notes in Artificial Intelligence*, pages 297–308. Springer-Verlag, Berlin, 1992.

[61] M. Gabbrielli and M. C. Meo. Fixpoint Semantics for Partial Computed Answer Substitutions and Call Patterns. In H. Kirchner and G. Levi, editors, *Algebraic and Logic Programming, Proceedings of the Third International Conference*, volume 632 of *Lecture Notes in Computer Science*, pages 84–99. Springer-Verlag, Berlin, 1992.

[62] H. Gaifman and E. Shapiro. Fully abstract compositional semantics for logic programs. In *Proc. Sixteenth Annual ACM Symp. on Principles of Programming Languages*, pages 134–142. ACM, 1989.

[63] H. Gaifman and E. Shapiro. Proof theory and semantics of logic programs. In *Proc. Fourth IEEE Symp. on Logic In Computer Science*, pages 50–62. IEEE Computer Society Press, 1989.

[64] M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programs. In R. A. Kowalski and K. A. Bowen, editors, *Proc. Fifth Int'l Conf. on Logic Programming*, pages 1070–1079. The MIT Press, Cambridge, Mass., 1988.

[65] R. Giacobazzi. *Semantic Aspects of Logic Program Analysis.* PhD thesis, Dipartimento di Informatica, Università di Pisa, 1992.

[66] R. Giacobazzi, S. K. Debray, and G. Levi. A Generalized Semantics for Constraint Logic Programs. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1992*, pages 581–591, 1992.

[67] R. Giacobazzi and L. Ricci. Pipeline Optimizations in AND-Parallelism by Abstract Interpretation. In D. H. D. Warren and P. Szeredi, editors, *Proc. Seventh Int'l Conf. on Logic Programming*, pages 291–305. The MIT Press, Cambridge, Mass., 1990.

[68] R. Giacobazzi and L. Ricci. Detecting Determinate Computations by a Bottom-up Abstract Interpretation. In B. Krieg-Brückner, editor, *Proceeedings ESOP '92*, volume 582 of *Lecture Notes in Computer Science*, pages 167–181. Springer-Verlag, Berlin, 1992.

[69] P. M. Hill and J. W. Lloyd. Analysis of meta-programs. In H. Abramson and M.H. Rogers, editors, *Meta-programming in Logic Programming*, pages 23–51. The MIT Press, Cambridge, Mass., 1989.

[70] D. Jacobs and A. Langen. Static Analysis of Logic Programs for Independent AND Parallelism. *Journal of Logic Programming*, 13(2 & 3):291–314, 1992.

[71] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *Proc. Fourteenth Annual ACM Symp. on Principles of Programming Languages*, pages 111–119. ACM, 1987.

[72] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. Technical report, Department of Computer Science, Monash University, June 1986.

[73] G. Janssens and M. Bruynooghe. Deriving descriptions of possible values of program variables by means of abstract interpretation. *Journal of Logic Programming*, 13(2 & 3):205–258, 1992.

[74] N. D. Jones and H. Søndergaard. A Semantics-based Framework for the Abstract Interpretation of Prolog. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, pages 123–142. Ellis Horwood Ltd, 1987.

[75] K. Kanchanasut and P. Stuckey. Transforming Normal Logic Programs to Constraint Logic Programs. *Theoretical Computer Science*, 105:27–56, 1992.

[76] T. Kawamura and T. Kanamori. Preservation of Stronger Equivalence in Unfold/Fold Logic Programming Transformation. In *Proc. Int'l Conf. on Fifth Generation Computer Systems*, pages 413–422. Institute for New Generation Computer Technology, Tokyo, 1988.

[77] R. Kemp and G. Ringwood. An Algebraic Framework for the Abstract Interpretation of Logic Programs. In S. K. Debray and M. Hermenegildo, editors, *Proc. North American Conf. on Logic Programming'90*, pages 506–520. The MIT Press, Cambridge, Mass., 1990.

[78] H. J. Komorowski. Partial evaluation as a means for inferencing data structures in an applicative language: A theory and implementation in the case of PROLOG. In *Ninth ACM Symp. on Principles of Programming Languages*, pages 255–267. ACM Press, 1982.

[79] G. Kreisel and J. L. Krivine. *Elements of Mathematical Logic (Model Theory).* North-Holland, Amsterdam, 1967.

[80] K. Kunen. Negation in logic programming. *Journal of Logic Programming*, 4:289–308, 1987.

[81] J.-L. Lassez and M. J. Maher. Closures and Fairness in the Semantics of Programming Logic. *Theoretical Computer Science*, 29:167–184, 1984.

[82] G. Levi. Models, Unfolding Rules and Fixpoint Semantics. In R. A. Kowalski and K. A. Bowen, editors, *Proc. Fifth Int'l Conf. on Logic Programming*, pages 1649–1665. The MIT Press, Cambridge, Mass., 1988.

[83] G. Levi and P. Mancarella. The Unfolding Semantics of Logic Programs. Technical Report TR-13/88, Dipartimento di Informatica, Università di Pisa, 1988.

[84] G. Levi, M. Martelli, and C. Palamidessi. Failure and success made symmetric. In S. K. Debray and M. Hermenegildo, editors, *Proc. North American Conf. on Logic Programming'90*, pages 3–22. The MIT Press, Cambridge, Mass., 1990.

[85] G. Levi and D. Ramundo. A formalization of metaprogramming for real. In D. S. Warren, editor, *Proc. Tenth Int'l Conf. on Logic Programming*, pages 354–373. The MIT Press, Cambridge, Mass., 1993.

[86] G. Levi and G. Sardu. Partial Evaluation of metaprograms in a multiple worlds logic language. *New Generation Computing*, 6:227–247, 1988.

[87] J. W. Lloyd. Declarative error diagnosis. *New Generation Computing*, 5(2):133–154, 1987.

[88] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1987. Second edition.

[89] J. W. Lloyd and J. C. Shepherdson. Partial Evaluation in Logic Programming. *Journal of Logic Programming*, 11:217–242, 1991.

[90] J. Lobo, J. Minker, and A. Rajasekar. *Foundations of Disjunctive Logic Programming*. The MIT Press, Cambridge, Mass., 1992.

[91] M. J. Maher. Equivalences of Logic Programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 627–658. Morgan Kaufmann, Los Altos, Ca., 1988.

[92] M. J. Maher and R. Ramakrishnan. Déjà Vu in Fixpoints of Logic Programs. In E. Lusk and R. Overbeek, editors, *Proc. North American Conf. on Logic Programming'89*, pages 963–980. The MIT Press, Cambridge, Mass., 1989.

[93] K. Marriott and H. Søndergaard. Bottom-up Abstract Interpretation of Logic Programs. In R. A. Kowalski and K. A. Bowen, editors, *Proc. Fifth Int'l Conf. on Logic Programming*, pages 733–748. The MIT Press, Cambridge, Mass., 1988.

[94] K. Marriott and H. Søndergaard. Semantics-based Dataflow Analysis of Logic Programs. In G. Ritter, editor, *Information Processing 89*. North-Holland, 1989.

[95] M. Martelli and C. Tricomi. A new SLDNF-tree. *Information Processing Letters*, 43(2):57–62, 1992.

[96] B. Martens and D. De Schreye. Why Untyped Meta-Programming is not (much of) a problem. Technical Report CW159, Katholieke Universiteit Leuven, Department of Computer Science, December 1992.

[97] A. Messora and M. Martelli. Declarative semantics of meta-logic predicates in logic programming. Technical report, Università di Genova, DISI, 1992.

[98] J. Minker and A. Rajasekar. A Fixpoint Semantics for Disjunctive Logic Programs. *Journal of Logic Programming*, 9:45–74, 1990.

[99] U. Nilsson. Abstract Interpretation: A Kind of Magic. In J. Maluszyński and M. Wirsing, editors, *Programming Language Implementation and Logic Programming, Proceedings 3rd Int'l Symposium PLILP'91*, volume 528 of *Lecture Notes in Computer Science*, pages 299–309. Springer-Verlag, Berlin, 1991.

[100] H. Rasiowa and R. Sikorski. *The Mathematics of Metamathematics.* North-Holland, Amsterdam, 1963.

[101] V. A. Saraswat. *Concurrent Constraint Programming Languages.* PhD thesis, Carnegie-Mellon University, January 1989.

[102] H. Seki. Unfold/fold transformation of stratified programs. In G. Levi and M. Martelli, editors, *Proc. Sixth Int'l Conf. on Logic Programming*, pages 554–568. The MIT Press, Cambridge, Mass., 1989.

[103] E. Y. Shapiro. *Algorithmic Program Debugging.* The MIT Press, Cambridge, Mass., 1983.

[104] H. Søndergaard. *Semantics-Based Analysis and Transformation of Logic Programs.* PhD thesis, The University of Melbourne, June 1990. Revised version of PhD thesis, University of Copenhagen, December 1989.

[105] P. J. Stuckey. Constructive Negation for Constraint Logic Programming. In *Proc. Sixth IEEE Symp. on Logic In Computer Science*, pages 328–339. IEEE Computer Society Press, 1991.

[106] A. Takeuchi and K. Furukawa. Partial evaluation of Prolog programs and its application to meta programming. In H.-J. Kugler, editor, *Information Processing 86*, pages 415–420. North-Holland, Amsterdam, 1986.

[107] H. Tamaki and T. Sato. Unfold/Fold Transformations of Logic Programs. In Sten-Åke Tärnlund, editor, *Proc. Second Int'l Conf. on Logic Programming*, pages 127–139, 1984.

[108] D. Turi. Extending S-Models to Logic Programs with Negation. In K. Furukawa, editor, *Proc. Eighth Int'l Conf. on Logic Programming*, pages 397–411. The MIT Press, Cambridge, Mass., 1991.

[109] M. H. van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, 1976.

[110] T. Vasak and J. Potter. Characterization of Terminating Logic Programs. In *Proc. Third IEEE Int'l Symp. on Logic Programming*, pages 140–147. IEEE Comp. Soc. Press, 1986.

[111] R. Venken. A PROLOG meta-interpreter for partial evaluation and its application to source-to-source transformation and query optimization. In T. O'Shea, editor, *ECAI-84: Advances in Artificial Intelligence*, pages 91–100. North-Holland, Amsterdam, 1984.

[112] L. Vieille. Recursive query processing: the power of logic. *Theoretical Computer Science*, 69:1–53, 1989.

[113] S. Yamasaki, M. Yoshida, and S. Doshita. A fixpoint semantics of Horn Sentences based on Substitution Sets. *Theoretical Computer Science*, 51:309–324, 1987.